Thesis for The Degree of Doctor of Philosophy

Understanding and Supporting Software Design in Model-Based Software Engineering

Rodi Jolak

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden, 2020

Understanding and Supporting Software Design in Model-Based Software Engineering

Rodi Jolak

ISBN 978-91-7833-746-0 (PRINT)
ISBN 978-91-7833-747-7 (PDF)

Technical Report No 181D
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden

Cover illustration: The upper part is adapted from Raphael's painting: the school of Athens. It shows Donato Bramante, as Euclid, surrounded by his students. The bottom part shows the process of design thinking.

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2020.

"To understand the universe at the deepest level, we need to know not only how the universe behaves, but why."

- Stephen Hawking, The Grand Design

Abstract

Context: Model-Based Software Engineering (MBSE) is a software development approach in which *models* play a central role and serve a multitude of purposes. On the one hand, models can be used as drafts for problem understanding, ideation, creative thinking, and communication. On the other hand, models can also be used as guidelines for documentation, implementation, and code-generation.

Motivation: Several studies claim that modeling in MBSE helps to improve software quality, productivity, and maintainability. However, modeling is frequently believed to be a time consuming approach that requires a lot of effort and often complicates matters. Generally, the entire field of Software Engineering (SE) perceives a discrepancy between empirical findings and practitioners' a priori beliefs, which are often based on personal perspectives on the development process.

Objective: We aim to address the interplay of belief and evidence in MBSE by contributing to the empirical understanding of software design. Moreover, we aim to support the design activities in MBSE in order to improve the overall process and achievement thereof.

Method: To achieve the objective of this research, we first conduct more in-depth empirical studies that investigate the socio-technical nature of MBSE. In particular, we observe practices and investigate challenges to MBSE in order to increase our understanding of this software engineering approach. Based on the results of our empirical investigations, we propose mechanisms and tools to support the design activities of MBSE.

Studies: We conduct multiple studies to understand and support software design in MBSE:

- We analyze the modeling process in order to understand how much effort is given to designing (i.e. thinking about the design of software systems), and how much effort is given to drawing the model (i.e. tool interaction).
- We also investigate development efforts and challenges in MBSE practices.
- Moreover, our endeavor to support the design activities of developers resulted in creating two software design environments: *OctoUML* and *OctoBubbles*. *OctoUML* is a collaborative software design environment that supports the mix of informal and formal notations. *OctoBubbles* is a multi-view interactive environment for concurrent visualization and synchronization of software design and code.

- Finally, we study the effect of geographic distance and software design representation on design collaboration and communication.

Results: By conducting the aforementioned studies, we find the following:

- We find that much of the modeling effort is devoted to designing (thinking and pondering of design decisions).
- Our inquiry into MBSE efforts shows that the most of MBSE effort is spent on collaboration and communication between developers. Moreover, we find that tool-related challenges are the most encountered challenges in MBSE. We uncover that specific tool-challenges are due to: (i) usability of the tools, (ii) the learning effort of the tool-chain, (iii) the interoperability of various tools, and iv) the installation and configuration of the tools.
- Multiple evaluations of *OctoUML* and *OctoBubbles* indicate a positive perception of the users regarding the usability of these environments.
- By studying the effect of distance on design collaboration, we find that co-located developers do and *actively* discuss more design decisions in the problem space than distributed developers.
- We also find that a graphical software design representation is better than a textual representation in promoting *active discussion* between developers and improving the *recall ability* of design details.

Suggestions: Our endeavour to understand and support the design activities in MBSE practices suggests that we can enhance MBSE processes by:

- Reducing the complexity of MBSE tool-support and enhancing the usability thereof. This would let developers spend more time on pondering and thinking of design decisions.
- Introducing explicit triggers for effective communication in co-located and distributed MBSE collaborations. This in turn would enhance the efficiency and effectiveness of the collaborative activities of the developers.
- Studying the social challenges in MBSE, and accounting for the effect of these challenges on team behavior and development activities.

Keywords: Software Engineering, Software Design, Software Modeling, MBSE Efforts and Challenges, Software Design Environments, Collaboration, Communication, Human Aspects, Empirical Studies.

Acknowledgment

I would like to express my sincere gratitude to a number of people who helped me in accomplishing my research goals. To my main supervisor Michel R.V. Chaudron for his continuous support and inspiring discussions. To my co-supervisors: Morten Fjeld and Eric Knauss for their collaboration, feedback, and interesting discussions. To my examiner Ulf Assarsson for his constructive feedback during the follow-up meetings. To professor Matthias Book for giving me helpful and constructive feedback on my licentiate thesis.

I would like to thank Professor André Van der Hoek for being the opponent of my PhD defence. Also, I would like to thank Professors Kurt Schneider, Juan de Lara, and Maryam Razavian for being a part of the grading committee.

Special thanks to Boban Vesin, Eric Umuhoza, Marco Brambilla, Truong Ho-Quang, Bilal Karasneh, Grischa Liebel, Dave Stikkolorum, Ramon Schiffelers, Duy Le Khanh, Kaan Sener, Andreas Wortmann, Bernhard Rumpe, Maxime Savary-Lelanc, Manuela Dalibor, Juraj Vincur, Ivan Polasek, Regina Hebig, Xavier Le Pallec, and Sebastien Gérard for their collaboration and contribution to the papers appended in this thesis. In particular, I would like to thank my office-mate *Truong* for his support and fruitful discussions.

I would like to thank my colleagues (researchers, PhD students, and admins) at the Software Engineering division for their support and help.

Finally, I would like to express my sincere gratitude to my family members for their love and continuous encouragement.

Rodi Jolak
Göteborg, 2020

List of Publications

Included publications

This thesis is based on the following publications:

(A) R. Jolak, E. Umuhoza, T. Ho-Quang, M.R.V. Chaudron, M. Brambilla "Dissecting Design Effort and Drawing Effort in UML Modeling"
In the 43th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 384-391. 2017.

(B) R. Jolak, B. Vesin, M.R.V. Chaudron "OctoUML: An Environment for Exploratory and Collaborative Software Design"
In the 39th International Conference on Software Engineering Companion (ICSE-C), pp. 7-10. 2017.

(C) R. Jolak, T. Ho-Quang, M.R.V. Chaudron, R.R.H. Schiffelers "Model-Based Software Engineering: A Multiple-Case Study on Challenges and Development Efforts"
In the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS), pp. 213-223. 2018.

(D) R. Jolak, K.D. Le, K.B. Sener, M.R.V. Chaudron "OctoBubbles: A Multi-view Interactive Environment for Concurrent Visualization and Synchronization of UML Models and Code"
In the 25th IEEE International Conference on Software Analysis, Evolution and Re-engineering (SANER), pp. 482-486. 2018.

(E) R. Jolak, A. Wortmann, M.R.V. Chaudron, B. Rumpe "Does Distance Still Matter? Revisiting Collaborative Distributed Software Design"
In IEEE Software Journal 35, no. 6: 40-47. 2018.

(F) R. Jolak, M. Savary-Lelanc, M. Dalibor, A. Wortmann, R. Hebig, J. Vincur, I. Polasek, X. Le Pallec, S. Gérard, M.R.V. Chaudron "Software Engineering Whispers: The Effect of Textual Vs. Graphical Software Design Descriptions on Software Design Communication"
In submission to EMSE Journal. Revised version submitted on Dec. 2019.

Other publications

The following publications were published during my PhD studies, or are currently in submission. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

Conference (Peer reviewed)

(a) B. Karasneh, R. Jolak, M.R.V. Chaudron "Using Examples for Teaching Software Design: An Experiment Using a Repository of UML Class Diagrams"
In 2015 Asia-Pacific Software Engineering Conference (APSEC). IEEE, pp. 261-268. 2015.

(b) R. Jolak, B. Vesin, M.R.V. Chaudron "Using Voice Commands for UML Modeling Support on Interactive Whiteboards: Insights and Experiences"
In Proceedings of the 20th Ibero American Conference on Software Engineering (CibSE) @ICSE17, pp. in print. 2017.

(c) R. Jolak "Understanding Software Design for Creating Better Design Environments"
Lic.Phil. Thesis. Chalmers University of Technology and Göteborg University, 111 p. 2017.

(d) M.R.V. Chaudron, A. Fernandez-Saez, R. Hebig, T. Ho-Quang, R. Jolak "Diversity in UML Modeling Explained: Observations, Classifications and Theorizations"
In International Conference on Current Trends in Theory and Practice of Informatics, pp. 47-66. 2018.

(e) B. Vesin, A. Klašnja-Milićević, K. Mangaroska, M. Ivanović, R. Jolak, D. Stikkolorum, M.R.V. Chaudron "Web-based educational ecosystem for automatization of teaching process and assessment of students"
In Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics, p. 33. ACM 2018.

(f) R. Jolak, A.Wortmann, G. Liebel, E. Umuhoza, M.R.V. Chaudron "The Design Thinking of Co-located vs. Distributed Software Developers: Distance Strikes Again!"
In submission to the International Conference on Global Software Engineering (ICGSE). 2020.

Workshop (Peer reviewed)

(a) M.R.V. Chaudron, R. Jolak "A Vision on a New Generation of Software Design Environments"
In First International Workshop on Human Factors in Modeling (HuFaMo 2015). CEUR-WS, pp. 11-16. 2015.

(b) R. Jolak, B. Vesin, M. Isaksson, M.R.V. Chaudron "Towards a New Generation of Software Design Environments: Supporting the Use of Informal and Formal Notations with OctoUML"
In Second International Workshop on Human Factors in Modeling (HuFaMo 2016). CEUR-WS, pp. 3-10. 2016.

(c) R. Jolak, G. Liebel "Position Paper: Distances and Knowledge Sharing in Collaborative Modeling"
In the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MoDELS-C), pp. 415-416. 2019.

Role in Research

In paper A ("*Dissecting Design Effort and Drawing Effort in UML Modeling*"), we conducted experiments in collaboration with researchers at the *Polytechnic University of Milan* in Italy and *Gadjah Mada University* in Indonesia. My contribution are the experiment design, effort estimation approach, data analysis and results discussion. In terms of paper writing, I did the majority of the writing. The co-authors contributed with reviews and the writing of the execution procedure of the experiments.

The work in paper B ("*OctoUML: An Environment for Exploratory and Collaborative Software Design*") is done in collaboration with Dr. Boban Vesin. The main contributions of this paper are the demonstration video of *OctoUML* and the discussion of the evaluations that I did in other publications. My supervisor and I designed *OctoUML*. The first version of *OctoUML* was mainly created by two B.Sc. students. I also contributed by developing different functionalities of *OctoUML*.

In paper C ("*Model-Based Software Engineering: A Multiple-Case Study on Challenges and Development Efforts*"), we conducted a multiple-case study in collaboration with researchers at Eindhoven University of Technology. My contributions are mainly data analysis and discussion of the results. In terms of paper writing, I did the majority of the writing.

OctoBubbles, which is described in paper D, is the result of a M.Sc. thesis that I supervised together with Prof. Michel Chaudron. I was involved in examining the related work, shaping the approach, planning the user study, and discussing the resutls. I did the majority of paper writing.

The work in paper E ("*Does Distance Still Matter? Revisiting Collaborative Distributed Software Design*") is done in collaboration with researchers at RWTH Aachen University. In particular, Dr. Andreas Wortmann and I did the majority of the work. We designed the *distributed* case study and analyzed both the *co-located* and *distributed* case studies. We also did the majority of paper writing.

In paper F ("*Software Engineering Whispers*"), we conducted a family of experiments in collaboration with three universities: RWTH Aachen University, University of Lille, and the Slovak University of Technology. My contributions are the experiments' design, coordination between the different universities, data analysis approach, and discussion of results. I did the majority of paper writing. Beside contributing with reviews, the co-authors contributed in writing different parts of the paper, such as the related work, approach, discussion of the results, and threats to the validity of the experiments.

Contents

Chapter 1

Introduction

Model-Based Software Engineering (MBSE) is an engineering approach for software development in which models play an important role, without necessarily being the key artifacts of the development [1]. This approach has been devised to handle complexity (by means of abstraction) and enhance productivity in software development [2].

One argument in discussions about the adoption of MBSE is its in-effectiveness, e.g., the supposedly large effort it takes to do modeling. In particular, many practitioners consider MBSE a time consuming approach that requires a lot of effort and often complicates matters [3]. However, several studies claim that MBSE helps to improve software quality, productivity and support maintainability (e.g., [4, 5]).

More in general, the whole field of software engineering perceives the discrepancy between scientifically validated results (e.g., in the empirical software engineering) and developers' beliefs, usually based only on personal experience of the development processes. For this reason, there is a need to address the interplay of belief and evidence in software engineering practices, because knowledge should be motivated by experience and observation rather than by intuition [6, 7]. Therefore, in this thesis we conduct empirical studies in order to gain knowledge and increase understanding by observing and evaluating MBSE processes, tools and developers' activities.

Based on the purpose of modeling in MBSE, we highlight in Figure 1.1 that models can be used as:

(a) *drafts* for ideation, design exploration, understanding, externalization of thoughts, creative thinking, sketching, prototyping, or communication of abstract high-level details of software structure and behavior. We characterize this way of modeling as *informal modeling* [8].

(b) *guidelines* for documentation, verification and validation, implementation, code-generation, or communication of the complete low-level details of

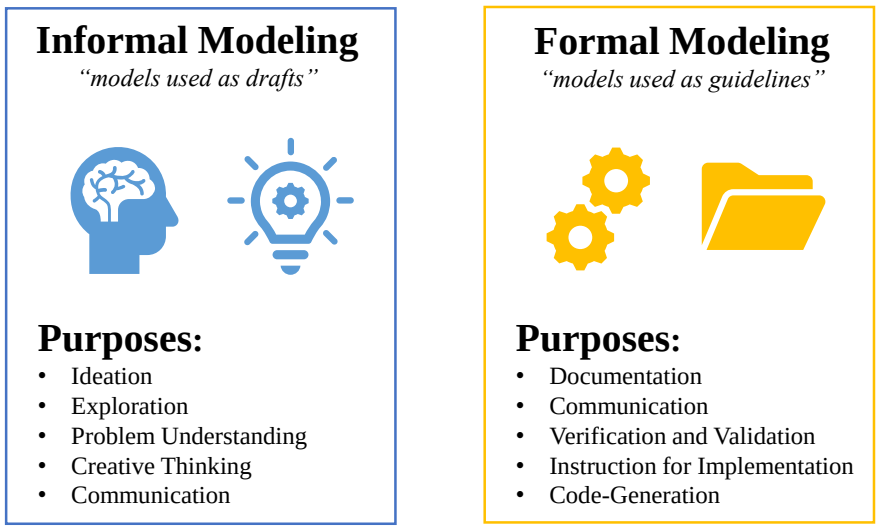


Figure 1.1: The purpose of using informal and formal modeling.

software structure and behavior. In this case, models play a central role and drive the development process. In contrast to informal modeling, we distinguish this approach as *formal modeling* [8].

Accordingly, we classify modeling tools into two categories: informal and formal [8]. On the one hand, we mean by informal tools any tool that supports informal modeling in the sense that it does not constrain the modeling notations that can be used by software designers. Examples of informal tools are *whiteboards* and *pen & paper*. These tools are flexible, easy to use and allow designers to unleash their expressiveness [9]. On the other hand, we mean by formal tools any Computer Aided Software Engineering (CASE) tool that supports the creation of models expressed through formal modeling notations.

Whiteboards, as informal modeling tools, do not serve well for persistence and transfer of designs. What typically happens in practice is that software designers go to the whiteboard, create a software design, take a picture of the created design, go back to their desks, run a CASE tool, and try to re-draw or formalize the design that they have previously created on the whiteboard. In this process, redrawing is an additional step using a separate tool and requiring extra effort. Moreover, the transition between informal (whiteboards) and formal tools (CASE tools) introduces a discontinuity that can be a source of errors. Thus, rationale, ideas and the logical basis for design decisions can be easily lost when moving from the whiteboard to the CASE tool [10].

A common weakness in formal tools is poor support of designing practices [10]. Indeed, these tools support one or few formal modeling notations, and

hence restrict designers' expressiveness. To make matters worse, the majority of CASE tools are not designed for user experience, and their usability leaves to be desired [11]. In fact, there are reports showing that CASE tools are perceived complex and difficult to use [12, 13]. Moreover, the complexity of CASE tools is considered to be a reason that adversely affect the adoption of model-based approaches [14].

Both informal and formal modeling have their advantages and disadvantages. Yet, neither serves all purposes in designing software in MBSE. Therefore, in this thesis we also study new forms of software modeling tools which could better serve the multitude of purposes.

The remainder of this chapter is structured as follows: Section 1.1 provides the motivation of this PhD research. Section 1.2 details the research goals and questions. Section 1.3 provides the background of this research. Section 1.4 discusses the related work. Section 1.5 details the research methodology. Section 1.6 presents a brief summary of the included papers and their contribution towards achieving the objective of this PhD research. Section 1.7 wraps up this chapter with recommendations and plan for future work.

1.1 Research Motivation

In this Section, we describe the *Challenges* (C) that motivate the endeavor of our research in *Understanding* (U) and *Supporting* (S) the software design activities in MBSE. The selection of these challenges is based on reading scientific literature in MBSE and personal observations from conferring with experienced software developers using MBSE or struggling with MBSE adoption. The challenges that we report in this section are mainly related to:

- lack of understanding of the development efforts in MBSE, such as design, modeling, collaboration, artifacts navigation, and communication efforts.
- MBSE tool-related challenges, such as poor tool-usability and lack of tool-support for an effective software design collaboration as well as software artifacts visualization and navigation.

Next, we describe the challenges that motivate the endeavor of our research in *understanding* software design activities.

C.U.1. Design and Modeling Efforts. Regardless of the reported benefits of model-based approaches in literature, e.g. [5, 15], the use of modeling is still debated in practice [16]. This is because modeling is believed as an unnecessary and superfluous activity that introduces extra effort in the process of software development [3]. It also believed that the benefits of modeling take place after a long-term [5, 16], in the sense that modeling introduces an initial overhead at

the beginning, whereas benefits, such as increased productivity [17] and better software maintenance [18], start to take place at late stages.

We consider that the benefits of modeling does not only take place after a long-term, but also directly at early stages. In particular, we consider that part of the modeling effort could be actually devoted to reasoning and thinking about the design solution, while other parts of the effort are dedicated to drawing the design solution (i.e., representing the solution via a modeling notation). Here the challenges are to:

- dissect the effort spent on design thinking and the effort spent on model drawing in software modeling.
- understand how to reduce the drawing effort (i.e., the modeling cost).

By confronting these challenges, we aim to increase practitioners' and researchers' understanding of the efforts of software modeling and design processes. We also provide evidence regarding the impact of adopting MBSE on software development process and its achievement. Moreover, by understanding how to reduce the drawing effort, we contribute to reducing the overall effort of MBSE which, in turn, would increase the productivity of MBSE.

C.U.2. Understanding the Challenges and Development Efforts of MBSE. MBSE aims to increase the abstraction level and promote the automation of the development process [3]. A recurring theme in discussions about the adoption of MBSE is its effectiveness. This is because there is a lack of empirical assessment of the processes and (tool-)use of MBSE in practice.

Although MBSE claims many potential benefits, e.g., gains in productivity, and maintainability [19–21], its adoption has been facing a number of challenges, such as poor tool-support [5] and developers' diverse perception of the benefits of MBSE [22]. On the one hand, MBSE is considered as beneficial after it has been applied effectively in several application sectors [4, 23]. On the other hand, MBSE is considered as a time-consuming and unproven approach that merely complicates matters [3]. Devanbu et al. [6] suggest that more in-depth studies that address the interplay of belief and evidence in software practices are needed. In MBSE, the main challenges are to:

- increase our understanding of the process and use of MBSE in practice.
- address the interplay of belief and evidence in MBSE practices.

By confronting these challenges, we aim to increase the empirical understanding of MBSE challenges and development efforts. TFurthermore, exposing MBSE challenges and development efforts would make them a candidate subject for research that are concerned with MBSE process improvement.

C.U.3. Distributed MBSE. Global software engineering (GSE) is the practice of engineering software systems across geographical, socio-cultural, and temporal boundaries [24]. Organizations engage in GSE to reduce development costs and take advantage of proximity to markets and customers [25]. However when globalized, software engineering becomes less effective. Indeed, these organizations often face numerous challenges, including poor quality of globally developed software [26].

In collaborative MBSE, geographic distance can lead to socio-technical challenges that potentially affect the way software is developed. This leads to the following challenges:

- increase our understanding of software design practice and cognition in both co-located and distributed MBSE.
- increase our understanding of the impact of distance on collaborative software design activities.

By understanding software design practice and cognition in collaborative MBSE, we aim to identifying challenges that needs to be addressed or further studied in order to support the process of software design, which in turn would contributes to enhancing the effectiveness and efficiency of MBSE.

C.U.4. Graphical vs. Textual Design Descriptions. By investigating MBSE development efforts, Jolak et al. [27] find that the effort on communication is actually more than all of the efforts that developers spent in any of the other MBSE activities. Accordingly, they decided to study communication in-depth to determine elements or criteria of its efficiency and effectiveness. To communicate software design decisions to other stakeholders, developers create graphical or textual descriptions of these decisions. Graphical descriptions encode and present knowledge differently from textual descriptions. Moreover, these two categories of knowledge representation are differently processed by the human mind, as stated by Moody [28]. Empirical evidence on how graphical descriptions affect developer's achievement and development productivity is still underwhelming, as reported by Hutchinson et al [5] and Meliá et al. [29]. Thus, our focus in to understand how different software design descriptions (i.e., graphical versus textual) influence software design communication. Such understanding might lead to achieving more effective software design communication, which in turn would help in reducing the total effort of software development activities.

Next, we describe the challenges that motivate the endeavor of our research in *supporting* software design activities.

C.S.1. Support of Informal and Formal Modeling. Informal modeling facilitates thinking and fosters ideation [30]. Moreover, it presents an intuitive

way to prototype and communicate thoughts [31]. Informal modeling supports the process of software design and serve developers to inspect and develop one design idea as well as reflect on some other alternatives [32]. In practice, software developers often go to the whiteboard to discuss requirements, explore domain problems and sketch design solutions [33]. The reason is that whiteboards are flexible mediums and at easy disposal. Furthermore, whiteboards allow informal modeling i.e., there are no restrictions on the type or formality of the modeling notations that can be used. However, whiteboards do not offer means for automated data analysis and maintenance. Thus, the sketched diagrams often need to be formalized and re-modeled again using a CASE tool. This transition between informal and formal tools introduces a discontinuity that can be a source of errors and often requires extra effort. Indeed, rationale, ideas and the logical basis for design solutions can be easily lost when moving from the whiteboard to the CASE tool [10]. Moreover, CASE tools provide environments where only one or few formal modeling notations are supported. This actually limits the expressiveness of designers, especially in early-phase software design.

Table 1.1 is based on [34], and describes some advantages of informal and formal modeling tools. Our focus is to provide a *'one stop'* environment capable of supporting both informal and formal modeling, while preserving the advantages of both informal and formal tools. Such an environment would support the designing and modeling processes which in practice often go hand-in-hand, and reduce the effort of MBSE.

Table 1.1: Advantages of informal and formal modeling tools

	Informal Tools	**Formal Tools**
Clarity		High clarity because of strict adherence to syntax
Flexibility	Caters for improvisation of notation	
Ease of continuous design	In tools based on digital editing, editing (move, resize, delete, undo, etc.) is easier than in sketch-based tools such as whiteboards.	
Ease of learning Notation		Formal syntax checking helps in learning the proper syntax
Intuitiveness of using tool	Very simple to use; but limited in functionality	More difficult to learn, but advanced functionalities supported
Collaboration	Multiple people collaborating on a shared design prefer to use informal representations [35]	
Integration	Absence of a formal syntax (and semantics) prohibits exchange of designs	Formal syntax allows a formal representation of the design that can be exchanged with other tools

C.S.2. Usability. CASE tools are criticized of being complex and difficult to use [12, 14]. This complexity of CASE tools hinders their adoption and often

costs companies extra effort and money for training and learning endeavor. In particular, the interaction with these tools is not always well-designed for a user experience, easy learning, and effective use [14]. As a consequence, CASE tools are often considered as barriers to the adoption of model-based approaches [36]. Our focus is on the challenges of:

- providing rich features in a simple and intuitive User Interface (UI).
- making CASE tools fit easily into users' activities, rather than forcing users to fit their activities into the dictates of the tools [37].

Providing ways to overcome these challenges could bring a significant impact to the effectiveness and efficiency of the MBSE approach.

C.S.3. Interaction Modalities. One key aspect of usability is the manner in which users interact with the system. The interaction with current CASE-tools takes place essentially by using the mouse and keyboard. However, in recent times interaction technologies such as touch, voice and gesture [38] have matured, and become commonly used as interaction modalities with software systems. Like [39], we believe that supporting multiple interaction modalities within software design environments would allow the users to switch to a better-suited modality for the execution of one particular task. Our focus is to:

- provide more intuitive and effective interaction with CASE tools.

By confronting this challenge, we aim to enhance the usability of MBSE tools as well as reduce the effort of interaction with these tools. This in turn would enhance the overall efficiency of the MBSE approach.

C.S.4. Collaboration. Software engineering is a collaborative activity. More often than not, multiple developers work together on creating a software design. Moreover, these developers often collaborate with other stakeholders to shape the structure and behavior of software systems. In [27], we show that the majority of MBSE effort is spent on communication and collaboration between developers on MBSE activities.

Tools supporting MBSE are often deployed on personal computers, and only one developer can effectively interact with the PC at any one time. This actually limits the collaboration between developers. As globally-distributed projects become the norm in Software Engineering [40], developers who are geographically distributed need effective tool-support to accommodate remote-collaborative design meetings [24]. Our focus is on the challenges of:

- providing effective and efficient support for co-located software design in MBSE.

- supporting distributed software design in MBSE, while preserving the natural, effortless kind of awareness and communication that happens in collocated settings.

By providing tool-support for both co-located and distributed collaborative software design, we contribute to improve the collaboration, communication, and coordination between and among software development teams [41].

C.S.5. Artifacts Visualization. During all stages of development, developers seek to understand both the design and the implementation of the software system, as well as the relation between these two artifacts. The process of understanding often requires developers to create a mental view of high-level descriptive domain artifacts (i.e., models) and low-level artifacts (i.e., code) [42,43]. CASE tools do not provide developers with multiple simultaneously-visible views of both low- and high level software artifacts. Consequently, developers often lose the big picture and spend unnecessary effort on navigation and locating artifacts of interest. Indeed, in scientific literature it is reported that around 35% of developers time is spent on software navigation [44], and around 60% of developers time is given to software understanding activities [45]. Thus, our focus is to:

- provide an interactive approach for bidirectional, smooth navigation between software models and their corresponding source code.
- support program comprehension and tackle the problem of software artifacts' navigation.

By assisting software comprehension and navigation, we aim to reducing the overall effort of MBSE, which in turn helps in increasing the MBSE productivity.

1.2 Research Focus

In this Section, we describe the research goals and present the research questions.

1.2.1 Research Goals

The overall objective of this PhD project is to *Understand and Support the Design Activities in Model-Based Software Engineering.* To achieve the objective of this research, we define the following goals:

- G1: to better understand the activities of software design in relation to different design-purposes, and
- G2: to propose and evaluate solutions for supporting the design activities of software developers.

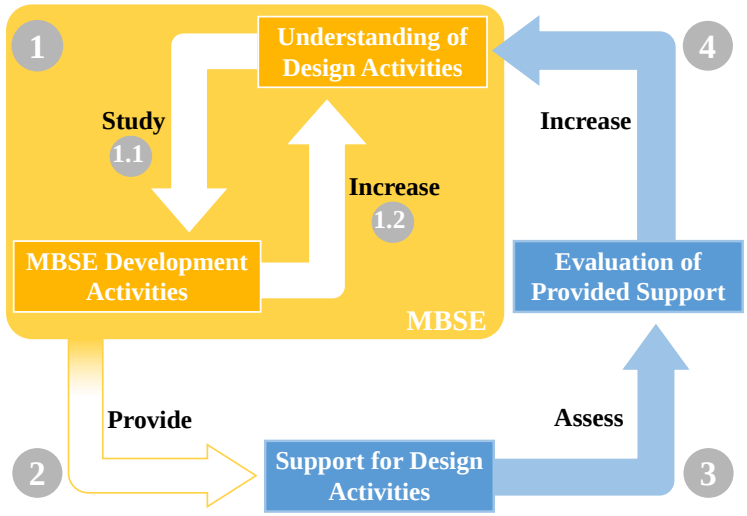


Figure 1.2: The approach to achieving the goals of this research.

Figure 1.2 shows the approach that we use to achieve the goals of this PhD research. In particular, we do the following steps:

1. We develop an understanding of the design activities of software developers by: (i) reviewing scientific literature in MBSE and (ii) pondering personal observations from conferring with experienced software developers using MBSE or struggling with MBSE adoption.

   1.1 Having reviewed the literature and pondered observations from discussions with experienced developers, we conduct empirical studies to study software design activities in the context of MBSE.

   1.2 Based on the results of the empirical studies, we increase our understanding of the design activities in MBSE.

2. Based on the results of the empirical investigations and our understanding of the design activities, we propose novel approaches to support development and design activities.

3. We empirically assess the proposed approaches.

4. Based on assessment results, we further increase our understanding of design activities in the MBSE domain.

1.2.2 Research Questions

To achieve our goals in understanding and supporting the design activities in MBSE, this PhD thesis addresses the following research questions:

- RQ1. How can the design effort and drawing (i.e., modeling and layouting) effort in software modeling be dissected?
  - RQ1.1. How much of the modeling effort is spent on designing?
  - RQ1.2. How much of the modeling effort is spent on drawing the design solution?

In this thesis, we propose that modeling tools in MBSE should support:

(F1) the mix of informal and formal modeling notations.

(F2) a smooth navigation between models and code.

Accordingly, we ask the following research question:

- RQ2. Can supporting (F1) and (F2) make the modeling tools of MBSE easier to use and more productive?
  - RQ2.1. Do developers find combining informal and formal modeling beneficial?
  - RQ2.2. Do developers find navigating between models and code beneficial?
- RQ3. How is the total effort spent on MBSE distributed over different development activities?
  - RQ3.1. How is the effort spent on different MBSE development activities distributed over time?
  - RQ3.2. How large is the portion of collaborative work in MBSE projects?
  - RQ3.3. What are the challenges that affect MSBE in practice?
- RQ4. How does distance influence the design activities of software developers?
  - RQ4.1. How does distance influence the amount of problem- and solution design decisions?
  - RQ4.2. How does distance influence design communication?
  - RQ4.3. What challenges are encountered when collaboratively designing software at a distance?
- RQ5. How does the representation of software design (graphical vs. textual) influence design communication?

Table 1.2 reports the research questions and links them to the targeted research goals. In particular, **RQ1**, **RQ3**, **RQ4**, **RQ5** and their sub-questions aim to achieve **Goal 1**. **RQ2** and its sub-question aim to achieve **Goal 2**.

Table 1.2: The research questions and the targeted research goals

PhD Goal	G1	G2
Description	To better understand the activities of software design in relation to different design-purposes	To propose and evaluate solutions for supporting the design activities of software developers
Research Question	RQ1, RQ3, RQ4, RQ5	RQ2

1.3 Background

In the first part of this section, we introduce the concepts of software design and software modeling. Later on, we provide details on existing modeling tools and their types.

1.3.1 Software Design

We distinguish between design as a process (i.e., designing) and as an artifact (i.e., final product of the design process). We define them as follows:

- **Definition** *Designing (process):* the process of thinking about, pondering over, making, shaping, and evaluating design decisions for something that is to be created.
- **Definition** *Design (artifact):* a set of design decisions that describes how something is to be built.

Ralph [46] describes design as a process that includes three primary activities: sense-making, co-evolution, and implementation. He defines these concepts as follows: Sense-making is the process where a design agent (an entity or group of entities that is capable of forming intentions and goals and taking actions to achieve those goals, and that specifies the structural properties of the design object) perceives its environment and the design object's environment and organizes these perceptions to create or refine the mental picture of context. Co-evolution is the process where the design agent simultaneously refines its mental picture of the design object based on its mental picture of context, and vice versa. Implementation is the process where the design agent generates or updates a design object using its mental picture of design object.

According to Budgen [47], the purpose of design is to produce a solution to a problem. The problem is basically described by means of the requirements

specification, and the solution is given via describing how the product should be constructed.

Dorst and Cross [48] describe the problem-solving aspect of design as a co-evolution of problem– and solution spaces. In particular, they describe design as the process of developing and refining together both the formulation of a problem and ideas for a solution, with constant iteration of analysis, synthesis, and evaluation processes between the problem space and solution space. Figure 1.3 shows the co-evolution of problem-solution during design as described by Dorst and Cross. Designers start by exploring the problem space and find a partial structure (P(t + 1)). That partial structure is then used to provide them with a partial structuring of the solution space (S(t + 1)). The designers consider the implications of the partial structure within the solution space, use it to generate some initial ideas for the form of a design concept, and so extend and develop the partial structuring (S(t + 2)). Some of this development of the partial structuring may be derived from references to earlier design projects. The designers transfer the developed partial solution structure back into the PS (P(t + 2)), and again consider implications and extend the structuring of the problem space. The ultimate goal is to create a matching problem–solution pair.

When designing software, software developers together with other stakeholders explore the interplay of problem and solution space. To handle problems during design, expert developers intuitively practice *design thinking* [49]. In *design thinking*, developers explore the problem and solution spaces separately, and iteratively align these two. Developers who are in the *design thinking* status are cognitively aware that there is a relation between the way they understand the problem space and the solution space, and any choice or decision they take at a certain time will impact the final results.

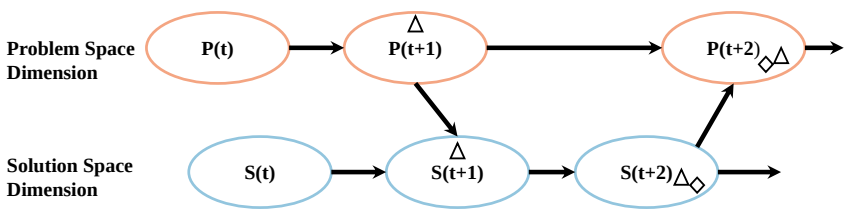


Figure 1.3: Co-evolution of problem-solution during design [48]

Lindberg et al. [50] highlight that *design thinking* fosters three main activities (see Figure 1.4):

(a) *Exploration of the problem space*: by analyzing the problem space and framing the design problem;

(b) *Exploration of the solution space*: by creatively devising and evaluating design solutions; and

(c) *Iterative alignment of both spaces*: by keeping the problem space in mind for refining and revising the chosen solutions.

Furthermore Lindberg et al. indicate that *design thinking* can broaden the problem understanding and problem solving capabilities in IT development processes. This is in line with Brooks [51], who considers *design thinking* an exciting new paradigm for dealing with problems in software and IT development.

In software engineering, Petre and Van der Hoek consider that *designing* is an activity that developers do throughout the entire development process, despite of their different roles in a project [33]. For example, developers do not only elicit requirements, they actually design the requirements by discussing and shaping these requirements with the contributing stakeholders. Developers also design their code by composing, analyzing, and evaluating algorithms to ensure, e.g., an efficient algorithm run-time. Similarly, developers design use cases, interactions, user interfaces, and test cases.

In this thesis, we look into design thinking as a *cognitive style* that developers adopt during problem solving [49]. Moreover, in this thesis we use the term *software developer* to indicate any person who takes a role in the software development process. In other words, we consider architects, designers, programmers, and testers as software developers. Like [33], we also consider that all developers (architects, designers, programmers, etc.) make design decisions.

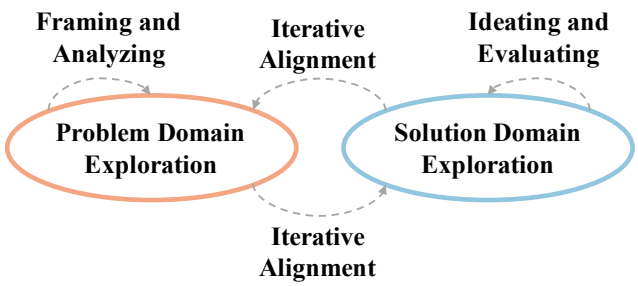


Figure 1.4: Problem solving with design thinking [50]

1.3.2 Software Modeling

We distinguish between *modeling* as a process and *model* as an artifact (i.e., final product of the modeling process):

- **Definition** *Modeling (process):* the process of creating a model by choosing what to represent and how to represent it.
- **Definition** *Model (artifact):* A model is a systematic and abstract representation of a concept.

In software engineering, models of software designs serve a multitude of purposes [52]:

- Planning: splitting the work in parts and delegate these to different teams or developers.
- Progress Monitoring: providing an overview of the progress of individual components, or by showing which components have been completed.
- Cost Estimation: providing a breakdown of the system into components.
- Risk Management: making explicit, which components are needed in a system, this in turn triggers discussion about possible risks that may arise in the construction and composition of components into the overall system.
- Compliance: verifying that the implementation indeed conforms to the design.
- Coordination: providing a common standard on how to handle the design and implementation.
- Knowledge Sharing: modeling a system is a way of capturing knowledge about a system. Through its representation this knowledge can be shared in a development team.
- Ideation: helping in inventing ideas and exploring new directions.
- Analysis: allowing various types of analysis of the system, ranging from more qualitative 'what if' scenarios (e.g., about maintainability) to quantitative analysis of extra-functional properties such as performance, reliability, safety, and others.
- Prototyping: models can be used to demonstrate and try out how the system will work.
- Code generation: models of the system are essential for code-generation. The main objective of this, is to enhance software productivity.

- Traceability: providing an intermediate abstraction, especially between requirements and the implementation. So, models can act as a pivot point and aid in establishing traceability between requirements and the implementation.

- Testing: models can be the basis for specifying and prioritizing tests.

A modeling language is a medium that lets developers *specify* the models for their systems. Modeling languages can be General Purpose (GPLs) or Domain Specific Languages (DSLs). GPLs can be applied to any domain. Examples of GPLs are: Petri nets [53] and the Unified Modeling Language (UML[1]). DSLs are designed specifically for a certain application domain or context. Examples of DSLs are: HTML[2] and SQL [54].

Software developers often use software modeling tools to prototype or document a software design. These modeling tools can be divided into two categories: informal tools and formal tools.

Informal tools support the creation of informal designs, such as elements and symbols –often sketchy– that do not adhere to a modeling language or syntax. Such tools are typically used for problem domain analysis, ideation, and solution exploration. A typical example of an informal tool is the whiteboard.Whiteboards are flexible tools and do not constraint the notation that can be used.

Formal modeling tools enforce the use of formally defined syntax of some modeling languages. Indeed, they typically provide support for the creation of one or more representations that adhere to one or more modeling languages, such as, e.g., the UML. Typical examples are CASE tools, such as Rational Rose, Enterprise Architect, Visual Paradigm, and StarUML. These tools are mainly used to describe or communicate the software system. Sometimes, these tools are used to create a detailed model that serves as a blueprint for implementing the software.

1.3.3 Relation between Software Design and Modeling

In practice, software design and modeling go hand-in hand. Figure 1.5 illustrates the relation between the software design and the software modeling process. We consider software design as a cognitive process [49] that happens inside of the mind of the developer. This process enables:

- the exploration of the problem space by analyzing and framing design problems,

- the exploration of the solution space by creatively ideating and evaluating design solutions, and

[1] http://www.omg.org/spec/UML
[2] https://www.w3.org/html/

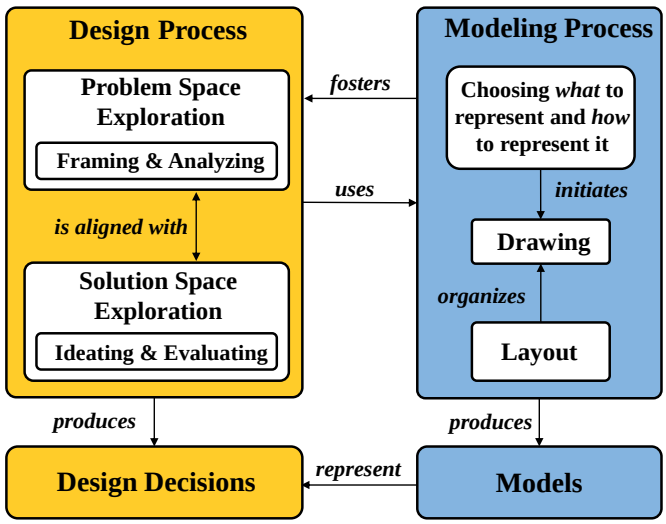


Figure 1.5: Software modeling and design go hand-in hand.

- the iterative alignment of both spaces by keeping the problem space in mind for refining and revising the chosen solutions.

As a result, the design process progressively produces a set of design decisions. Software modeling is often used to externalize or express the design decisions that happen inside of the developer's mind. In this process, developers create (i.e., draw) and organize (i.e., layout) models by choosing what to represent (e.g., the decomposition of a system in subsystems) and how to represent it (e.g., a component diagram). Software models provide means for representing software design decisions on different levels of abstraction and from multiple perspectives [55]. In MBSE, developers iteratively engage in the modeling and design process. This is because models provide visual representations of the design which foster the design process [56] by triggering analysis, ideation, and evaluation of design decisions in both the problem– and solution space.

1.3.4 Model-Based Software Engineering

Model-Based Software Engineering (MBSE) is an engineering approach that aims to handle complexity and increase the efficiency in the development of software [2]. This engineering approach addresses complexity by means of abstraction and modeling. Models in MBSE play an important role although they are not necessarily the key artifacts of the development (i.e., they do not drive the process) [1]. MBSE is used in industry [5, 15], and several

empirical studies show benefits of MBSE, e.g., increased software productivity and improved quality [5, 57]. From these papers, it seems that most increments in software productivity are actually obtained from increasing the abstraction level of the artefacts that represent the software system.

There are versions of model-based approaches adopted in practice such as, e.g., Model-Driven Software Enginnering (MDSE). Compared to MBSE, MDSE is a model-centric paradigm where models are the main artifacts of software development [58]. Documents such as code and test cases can be generated automatically from the models used to develop the system.

MBSE is a super-set of MDSE (See figure 1.6). In MBSE, developers can create models of the system during the the analysis and design phase and subsequently use these models as blueprints to manually write the code. Also, developers can automatically generate code from formal models. Models in MBSE still play an important role but are not the central artifacts of the development process and may be less complete (i.e., they can be used more as blueprints or sketches of the system) than those in an MDSE approach. Indeed, MBSE is a softer version of MDSE [1]. In MBSE, the audience of models are primarily humans and secondarily machines. In contrast, the audience of models in MDSE are primarily machines and secondarily humans.

In MBSE the details and abstraction-level are scalable, whereas the aim of MDSE to generate code, requires that models are complete, consistent and adhere strictly to formal standards for syntax.

The software design in MBSE approaches is represented by means of software models. In other *non* model-based software engineering approaches, software design is often represented by means of natural language or informal sketches.

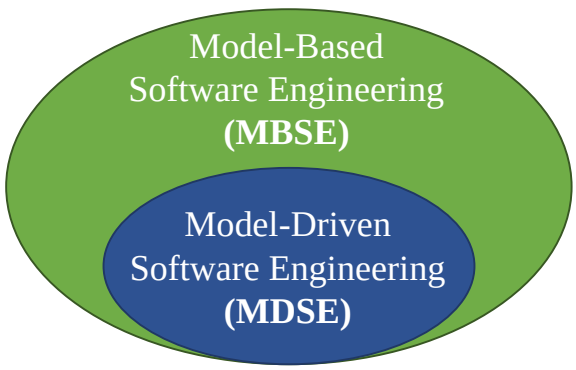


Figure 1.6: Relationship between MBSE and MDSE [1]

1.4 Related Work

In this section we describe some related work that focuses on understanding- and supporting software design.

1.4.1 Understanding Software Design

Petre and Van Der Hoek [59] organized a workshop to study professional software designers in action. They provided video-recordings and transcripts of three two-person teams who were assigned to create a software design for the same set of requirements on a whiteboard. Over fifty researchers were invited to participate in the workshop. They were asked to explore the videos to understand design strategies and types of activities that professional designers engage in during software design sessions. In the following paragraphs, we report the findings of some of the researches who took a part in the workshop:

- Tang et al. [60] tried to understand how software designers do design reasoning and decisions, and how the decision making process influences the effectiveness of the design process. They found that planning design discussions in an opportune way improves the exploration of design solutions. They also found that the manners by which the decisions are made have an impact on the use of design time and derived solutions. Furthermore, it seems that the use of design reasoning techniques contributes to the effectiveness of software design.

- Sharif et al. [61] analyzed the video-recordings and explored the software design strategies and activities that happened in each design session. They identified some of the time-consuming design activities such as decisions about the logic, discussion of uses cases, drawing class diagrams, and drawing the user interface. Furthermore, they found that planning and a high degree of agreement between designers plays a significant role in delivering a detailed design that covers most of the requirements.

- Baker and Van Der Hoek [62] studied the video recordings in order to understand how software designers address software design problems. In particular, they evaluated the design sessions in terms of the discussed subjects, generated ideas and design cycles. They found that the design sessions were highly incremental and designers repeatedly returned to high level subjects, as well as to previously discussed ideas.

- Budgen [10] tried to observe the design sessions and identify principles for designing software design environments. He found that: (i) informal diagrams and notations are often used, (ii) there was a frequent switching between viewpoints and sections of the whiteboard, and (iii) the whiteboard annotations were chiefly performed by one person.

Razavian et al. [63] consider software design as a problem solving exercise. They theorize that software design thinking requires two minds: a *reasoning* mind that focuses on the process of logical design reasoning and a *reflective thinking* mind that challenges the reasoning mind by asking reflective questions. Razavian et al. conduct multiple case studies to understand how reflections on reasoning and judgments influence software design thinking. They find that reflection improves the quality of software design discourse which, in turn, is considered as a foundation for a good design [64].

Karis et al. [65] performed studies of remote collaboration at Google. They found that the use of videoconferencing and video portals contributes to the success of remote collaboration by:

- providing presence and status information,
- helping to establish mutual trust and common ground, and
- preventing misunderstandings.

However, when it comes to remote design collaboration, Karis and his colleagues highlighted that developers at Google found collaboration over videoconferencing and video portals a pale imitation of face-to-face interaction. Moreover, the developers complained that the video portals at Google lacked a shared drawing tool to facilitate sketching, designing, and brainstorming.

Heijstek et al. measure how well participants extract design decisions from different software architecture representations [66]. The researchers collected participant-specific information in two questionnaires, filmed participants during tasks, and asked them to think out loud. The experiment comprised of four architectures, out of which each consisted of a graphical and a textual description. Participants (students and professional developers) were asked three questions per architecture. The authors observed that no notation was clearly superior in communicating architecture design decisions. Nonetheless, participants tended to first look at the graphical notation before reading the text. The authors attribute this to the clarity of the graphic representation, which enables participants to grasp the structure of the model more quickly.

1.4.2 Supporting Software Design and MBSE

On the one hand, several studies point to the lack of adequate and easy-to-use tools for supporting the activities of software developers in MBSE.

By surveying 155 Italian software professionals, Torchiano et al. [18] considered the lack of competencies and supporting tools as the main show stoppers preventing the adoption of modeling and model-driven techniques.

Mussbacher et al. [22] reflected on the opinions of 15 MDE experts on the biggest problems with MDE technologies over the last 20 years. The authors highlighted that tools *usability* and *adoption*, people's diverse perception of

MDE, inconsistencies between software artifacts, and lack of fundamentals in MDE are considered as hindrances to MBE adoption.

In the embedded systems domain, Liebel et al. [17] analyzed survey-responses from 122 professionals working with MBE, and considered that interoperability between (MBE/MDE) tools as a main challenge to MBE adoption. Moreover, other factors such as, high effort to train developers and tools (poor) usability, were also identified as secondary MBE challenges.

Whittle et al. [14] explored tool-related issues in affecting the adoption of MDE in industry. Based on interviews conducted with twenty companies, they observed that the interviewees emphasized tool immaturity, complexity, and lack of usability as major barriers against the adoption of MDE. Usability issues were reported to be related to the complexity of user interfaces. There is a lack of consideration of how people work and think. The authors suggested to match modeling tools to people, not the other way around, by producing more useful and usable tools, as well as supporting early-phase design and creativity in modeling.

In his paper "The cobblers' children", Budgen [10] provided some recommendations for future design support tools. In particular, informal diagrams and lists need to be integrated with other more formal notations. Ideally, a tool should be simple and support the transition from informal notations to formal notations. Budgen also stated that much of reasoning and rationale, during early-phase design, would quickly be lost unless it transformed into more formal description.

On the other hand, several studies propose and investigate approaches to support the activities of software developers in MBSE.

Dekel and Herbsleb [67] observed collocated object-oriented design collaborations and focused on representation use. They found that teams intentionally improvise representations and sketch informal diagrams in varied representations that often diverge from formal notations or languages. To support collaborative software design, they suggest that collaborative software design environments should focus on preserving contextual information, while allowing unconstrained mixing and improvising of notations.

Damm et al. [68] conducted user studies in order to understand the practice of software modelling. They observed that designers alternate between whiteboards and CASE tools, extend the semantics of the notations to support the design activities and allow expressiveness, sketch new ideas informally, and actively collaborate when they work in teams. Damm et al. developed a tool called *Knight*. *Knight* supports informal and formal modelling using gestures on an electronic whiteboard. In order to achieve intuitive interaction, *Knight* uses composite gestures and eager recognition of hand-drawn elements.

Chen et al. [69] developed *SUMLOW*, a sketching-based UML design tool for electronic whiteboard technology. *SUMLOW* allows the preservation of hand-drawn diagrams and supports the manipulation of them using pen-based

actions. It also allows the transformation of UML sketches into computer-drawn UML notations.

Grundy and Hosking [70] stated that software engineers often use hand-drawn diagrams as preliminary design artifacts and as annotations during design reviews. They created *MaramaSketch* to support flexible sketch-based input of diagrams. *MaramaSketch* enables the formalization of sketches into computer-drawn content.

Mangano et al. [71] identified some behaviors that occur during informal early-phase design. In particular, designers sketch different kinds of diagrams (e.g. box and arrow diagrams, UI mockups, flowcharts, etc.) and use impromptu notations in their designs. The authors implemented an interactive whiteboard system (called *Calico*) to support these behaviors, and identified some ways where interactive whiteboards can enable designers to work more effectively.

Wüest et al. [72] stated that software engineers often use paper and pencil to sketch ideas when gathering requirements from stakeholders, but such sketches on paper often need to be modeled again for further processing. A tool, *FlexiSketch*, was prototyped by them to support informal modeling of software requirements. *FlexiSketch* combines free-form sketching with the ability to annotate the sketches interactively for an incremental transformation into semi-formal models.

Al Abed et al. [73] stated that MDE faces several challenges which prevent its adoption. Examples of these challenges are the scalability and re-usability of models. The authors present *TouchRAM*, a multitouch-enabled tool for agile software design modeling which aims at developing scalable and reusable software design models. The tool exploits model interfaces and aspect-oriented model weaving to enable the designer to rapidly apply reusable design concerns within the design model of the software under development. The user interface of *TouchRAM* interface exploits mouse and touch-based input to enable intuitive and fast model editing. *TouchRAM* uses "eager" recognition of hand-drawn elements into formal modeling notations.

Brieler and Minas [74] stated that software developers tend to sketch during early phase software design, as sketching is more natural than using traditional software. They present an approach, called *DiaGen* to generate diagram editors based on language specification. Their approach relies on syntactical and semantical analysis of the sketches to resolve any ambiguities arising from the impreciseness of the hand-drawn diagrams. A recognition mechanism then transforms these sketched diagrams into formal diagrams which can be easily edited and further processed.

Baltes et al. [75] stated that informal sketches and diagrams are often detached from the source code they document. The authors conducted a study to understand the use of sketches in software engineering. They found that sketches are considered helpful to understand the related source code artifacts. For this, the authors proposed *SketchLink* to let software developers easily

capture, annotate, and link their diagrams and sketches to the correspondent source code artifacts.

In this thesis we present two tools, called *OctoUML* and *OctoBubbles*, that support the design and modeling processes in MBSE. These tools aim to:

- increase the productivity of MBSE, and
- enhance the usability of MBSE tools.

OctoUML bridges the gap between early-phase software design process (when developers often reason about the design and sketch their ideas) and the formalization and documentation process (when CASE tools are usually used to document the designs). *OctoBubbles* builds on *OctoUML*. It supports the navigation between software models and their corresponding (generated) source code. Table 1.3 shows the differences between our tools and the related work.

Table 1.3: Comparison between our tools and related work

		Modeling Purpose in MBSE				
		Exploration	Ideation	CC	Documentation	Code Generation
Tool	Knight	✓	✓	DS	✓	
	SUMLOW	✓	✓	DS	✓	
	MaramaSketch	✓	✓	DS	✓	
	Calico	✓	✓	RI & DS	IF	
	FlexiSketch	✓	✓	RI & DS	IF	
	TouchRam	✓	✓	MI & DS	✓	
	DiaGen	✓	✓	DS	✓	
	Our tools	✓	✓	✓	✓	✓

RI: Remote Interaction
MI: Multi-user Interaction
CC: Collaboration & Communication
DS: Design Sharing
IF: Informal Notations

In contrast to the related work, our tools aim to support the design, modeling, and models-code navigation processes in MBSE. Therefore, our tools support the following modeling purposes in MBSE:

- *Exploration* and *Ideation*, by supporting informal modeling notations.
- *Collaboration* and *Communication*, by supporting multi-user as well as remote interaction, version management, and design sharing.

- *Documentation*, by supporting transition from informal to formal modeling notations and saving design as XMI.
- *Code Generation*, by supporting the synchronization and navigation between software models and code.

More details on *OctoUML* and *OctoBubbles* are provided in Chapter 3 and Chapter 5.

1.5 Research Methodology

We employ design science and empirical research methods to achieve the goals of this research. With Goal 1, we aim to gain knowledge and increase understanding of software design activities. Hence, this is best addressed by conducting empirical software engineering research. With Goal 2, we aim to support the design activities of software developers. This can be addressed by design science research where new artifacts are created and evaluated in an iterative manner.

1.5.1 Empirical Research

The aim of empirical research is to gain knowledge and increase understanding by observing and evaluating processes, software tools and human-based activities [76]. To achieve Goal 1, we used *Controlled Experiments* and *Case Studies*.

1.5.1.1 Controlled Experiments

Controlled experiments help to investigate a testable hypotheses where one or more independent variables are manipulated to measure their effect on one or more dependent variables [77]. Thus, they are used to determine in precise terms whether a cause-effect relationship exists between the variables.

In the context of this research, we conduct a controlled experiment in Chapter 2 to dissect the design effort and drawing effort in UML modeling. In Chapter 7 we also conduct a controlled experiment to understand the effect of software design representation on design communication. The experiments are controlled in order to limit variables other than the chosen independent variables form affecting the results.

1.5.1.2 Case Studies

Yin defines case studies as empirical inquiries to perform a deep investigation of a particular phenomenon, where the boundary between the phenomenon and its real-life context cannot be clearly specified [78].

We conduct multiple-case studies in Chapters 4 and 6. In Chapter 4 we investigate MBSE challenges and development effort. In Chapter 6 we investigate the effect of distance on collaborative software design.

These studies are *exploratory- inductive* empirical research [79] to identify patterns in observations, seek new insights, and generate ideas and hypotheses for new research.

1.5.2 Design Science

Design science methodology is defined as the design and empirical investigation of artifacts in a given context [80]. In particular, it is an iterative process in which researchers engage in several cycles of two main activities: development of solutions (e.g., software prototyping), and evaluation of the proposed solutions in a given context to figure out whether the solutions effectively accommodate the problem.

To achieve Goal 2, we used a development approach which follows the paradigm of design science. In particular, we followed four main activities (See Figure 1.7):

- identifying needs and establishing requirements,
- developing alternative designs,
- building a prototype of the system, and
- evaluating the developed prototype.

We used the design science in Chapters 3 and 5. In Chapter 3 we develop and evaluate a software design environment, called *OctoUML*. *OctoUML* aims to support exploratory and collaborative software design. In Chapter 5 we develop and evaluate a second version of *OctoUML*, called *OctoBubbles*. *OctoBubbles* aims to facilitate software comprehension and navigation.

To evaluate the developed prototypes, we conduct *User Studies*. User studies offer a scientifically sound method to evaluate the strengths and weaknesses of different visualization and interaction techniques, as well as to investigate social and cognitive processes surrounding them [81]. In brief, these studies are based on collecting both quantitative and qualitative data based on standardized questionnaires that collect perceptions on the usability, efficiency and effectiveness of the developed prototypes.

1.6 Contributions

In this section, we summarize the main contributions of the six papers on which this PhD thesis builds. Figure 1.8 provides an overview of these papers and how they contribute to the main goals and research questions of the PhD

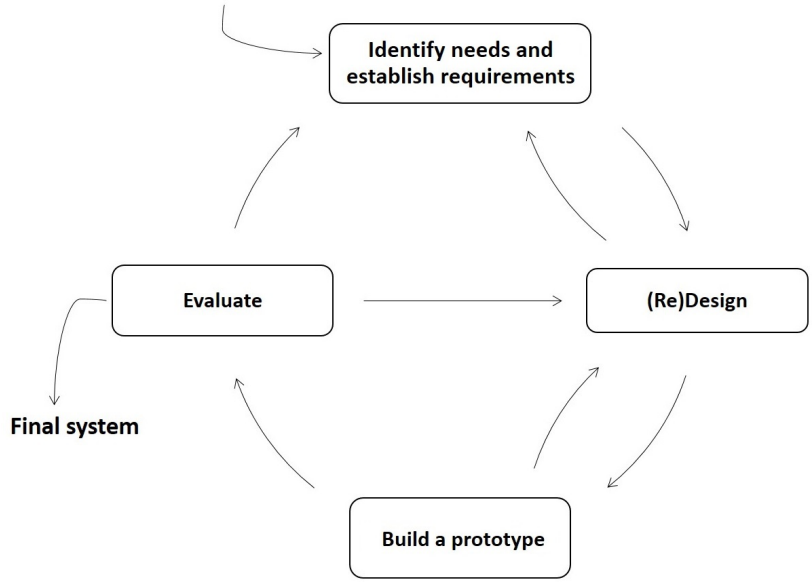


Figure 1.7: A simple model of the design science life-cycle

thesis. The included papers are presented in a rectangle shape and given an identification letter as follows:

- **Paper A:** Dissecting Design Effort and Drawing Effort in UML Modeling.
- **Paper B:** OctoUML: An Environment for Exploratory and Collaborative Software Design.
- **Paper C:** Model-Based Software Engineering: A Multiple-Case Study on Challenges and Development Efforts.
- **Paper D:** OctoBubbles: A Multi-view Interactive Environment for Concurrent Visualization and Synchronization of UML Models and Code.
- **Paper E:** Does Distance Still Matter? Revisiting Collaborative Distributed Software Design.
- **Paper F:** Software Engineering Whispers: The Effect of Textual Vs. Graphical Software Design Descriptions on Software Design Communication.

In Figure 1.8, the research questions (RQ) are represented by a round shape. The papers and research questions are placed on one or two horizontal layers corresponding to the PhD goals (Goal 1 and Goal 2) that they contribute to.

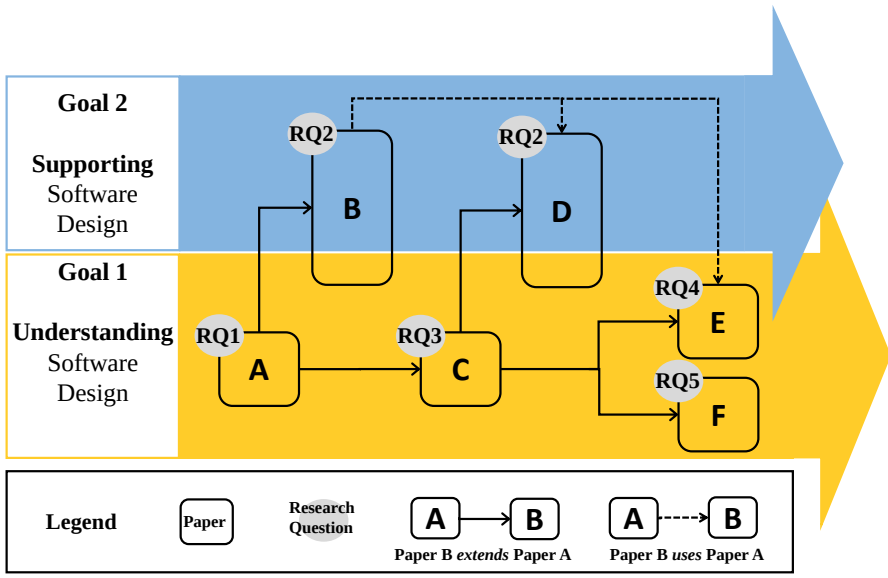


Figure 1.8: The research goals and questions of this PhD research.

The yellow layer represents Goal 1, *understanding software design*. The blue layer represents Goal 2, *supporting software design*. Papers that span over the yellow and blue arrows contribute to the two goals, e.g., Paper B contributes to both of these two goals of this PhD thesis. The solid and dashed black arrows between the rectangles (i.e., papers) indicate the direction in which the papers are built and extended. In particular, the paper at the head-end of the *solid* black arrow *extends* the paper at the tail-end. Also, the paper at the head-end of the *dashed* black arrow *uses* the paper at the tail-end.

Figure 1.9 provides an overview of the research context and contribution of this thesis. The research context is MBSE, a software development approach in which models play a central role. To achieve the objective of this research, we contribute to the empirical understanding of software design and modeling in MBSE. We investigate the effect of distance, social, and cognitive aspects on software design and modeling. We also investigate the effect of design representation on communication. Moreover, we create two software design environments: (i) *OctoUML* which enables distributed collaboration and supports the integration of informal and formal modeling notations, and (ii) *OctoBubbles* which supports the navigation between software models and code. In the next sections, we provide more details by describing the contribution of each paper included in this thesis.

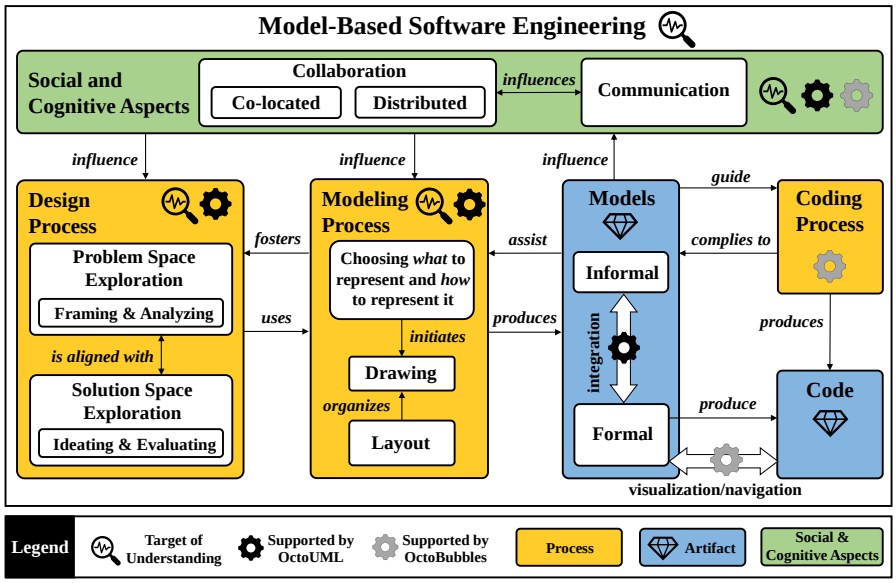


Figure 1.9: Overview of the research context and contributions.

1.6.1 Paper A: Dissecting Design Effort and Drawing Effort in UML Modeling

Model-based approaches are considered to be beneficial for improving software development productivity and product quality (e.g., [5]). However, these approaches are criticized because of the supposedly large effort they require to create and maintain models. Regardless of its reported benefits, software modeling is still believed as an unnecessary approach by some developers [82].

In this paper, we analyze the process of software modeling in order to find:

- how much effort is spent on the design of a solution (i.e., thinking and making design decisions), and
- how much effort is spent on drawing of the design solution with a modeling tool (i.e., tool interaction).

In particular, we consider that the modeling process consists of three main cognitive sub-activities:

- *Design*: ideation and thinking about the design.
- *Notation Expression*: representation of a design via the modeling notation.
- *Layout*: spatial organization of the elements of a model.

The drawing effort is the part of the modeling effort that is spent on *notation expression* and *layout*. The design effort is the part that is spent on ideation and thinking.

In paper A, we address the following research question:

- **RQ1.** How can the design effort and drawing effort in software modeling be dissected?
  - **RQ1.1.** How much of the modeling effort is spent on design?
  - **RQ1.2.** How much of the modeling effort is spent on drawing the design solution?

To compute the effort spent in each sub-activity, we conduct two-phase experiments. In the first phase, we measure the effort required to make the initial model of a system. This effort is the sum of the design, notation expression, and layout efforts. In the second phase, we measure the effort required to recreate the same model again, simply by redrawing the already defined solution (i.e., copying effort). At the end, we calculate the efforts for design, notation expression, and layout by assessing the time difference between the two phases.

The initial findings suggest that the majority of the modeling effort is devoted to design (i.e., design effort). This means that projects that create models incur at least significant thinking about the design. Moreover, we argue that the effort spent on using modeling tools (i.e., drawing effort) could be reduced by investigating better modeling-tool support. This is what we actually investigate in Paper B [83].

Chapter 2 provides more details on the design of the controlled experiments for this study, including the approach that is used to calculate the effort of each modeling sub-activity.

1.6.2 Paper B: OctoUML: An Environment for Exploratory and Collaborative Software Design

We briefly report the challenges that motivates the creation of *OctoUML*:

- *Integrating informal and formal notations*: The main challenge is to provide a *'one stop'* environment capable of supporting both informal and formal modeling, while preserving the advantages of both informal and formal tools [84]. Such an environment would support the designing and modeling processes which in practice often go hand-in-hand, and reduce the effort of MBSE.

- *Supporting multiple modes of interaction*: The main challenge is to provide more intuitive and effective interaction with CASE tools [38]. By confronting this challenge, we contribute to enhancing the usability of

MBSE tools as well as reducing the effort of interaction with these tools. This in turn would enhance the overall efficiency of the MBSE approach.

- *Supporting collaborative software design*: The challenge is to provide effective and efficient support for co-located software design sessions, and support distributed software design, while preserving the natural, effortless kind of awareness and communication that happens in collocated settings [24]. By providing tool-support for both co-located and distributed collaborative software design, we contribute to improve the collaboration, communication, and coordination between and among software development teams.

- *Usability*: The challenge is to provide rich features in a simple and intuitive User Interface (UI), and make CASE tools fit easily into users' activities, rather than forcing users to fit their activities into the dictates of the tools [37]. Providing ways to overcome these challenges could bring a significant impact to the effectiveness and efficiency of the MBSE approach.

Based on these motivations, we designed and created a new generation software design environment, called *OctoUML*. The design and implementation of *OctoUML* was challenging as we wanted to provide an environment that preserves the advantages of both informal and formal tools. One of the implementation challenges was enabling *OctoUML* to support multi-touch. According to the literature, it is reported that quiet often two or more people are involved in sketching when the whiteboard is used as a medium [35]. Multi-touch is an interaction technique that permits the manipulation of graphical entities with several fingers at the same time. This option allows concurrent collaborative modelling. In particular, it enables two or more developers to simultaneously work on the same canvas of the same device, especially when the device is an interactive whiteboard or a large touch screen. Another *OctoUML* implementation challenge emerged from deciding to equip *OctoUML* with a layering mechanism. In particular, the software design solution is part of one layer which we call the formal layer. While another layer, the informal layer, contains the informal sketchy elements e.g., hand-written comments, illustrative drawings, highlighting arrows or circles, etc. The user can then select to see combined layers or layers in isolation. A key advantage of such layers is that they allow the isolation of informal and formal elements. As a consequence, designers will be able to move and edit the content of each layer independently without disturbing the rest of the design. For instance, users might want to archive, print, or share the formal designs without including the sketchy elements. In that case, the formal layer can be a solution for them. On the other hand, having the two layers combined could help reveal some existing ambiguities in diagrams as well as give more insights to increase one's understanding of concepts, mainly, during diagram reviewing cycles.

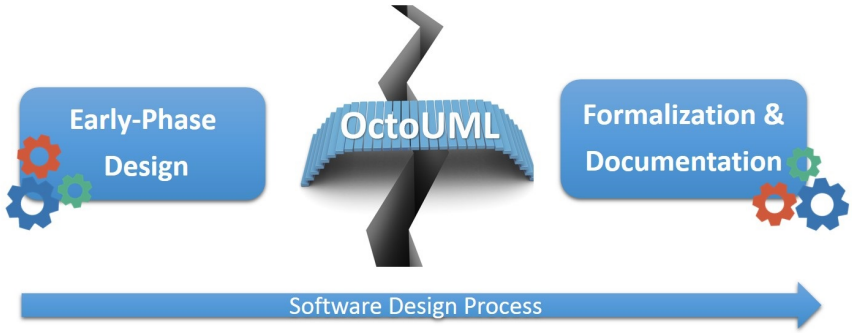


Figure 1.10: The main goal of *OctoUML*

Figure 1.10 shows the main goal of *OctoUML*. In particular, the goal is to bridge the gap between early-phase software design process (when developers often reason about the design and sketch their ideas) and the formalization and documentation process (when CASE tools are usually used to document the designs).

In this paper, we provide a link to a demonstration video which was specially recorded to disseminate the concept of *OctoUML*. A demonstration is an effective way to show the idea and functionalities of a system in action in order to enable people to fully grasp its value and potential. Moreover, we present the architecture and functionalities of *OctoUML*, and describe the design process by means of a design scenario. We further present the results of the evaluations performed in other papers (see the *Other Publications* in the list of publications of this thesis). The demonstration video of *OctoUML* can be consulted via the following link: `https://youtu.be/fsN3rfEAYHw`

OctoUML combines the advantages of both whiteboards and CASE tools, and supports the achievement of informal and formal modeling purposes (see Table 1.4). The key innovations of *OctoUML* are:

- enabling users to create and mix both informal hand-drawn sketches and formal computer-drawn notations at the same time on the same canvas.
- providing a selective recognition mechanism that is used to transform hand-drawn sketches into formalized contents.
- enabling of multi-user support on a single input device.

Paper B extends Paper A [85], where we argue that the effort of using modeling tools could be reduced by investigating better tool-support. The contributions of this paper are two-fold: design and creation of a new generation software design environment, *OctoUML*, and a qualitative evaluation and accompanying usability and efficiency discussion of the environment.

Table 1.4: Informal and formal modeling purposes as supported by the functionalities of OctoUML

		Modeling Purpose					
		Ideation	Exploration	Collaboration	Communication	Documentation	Code Generation
Functionality	Informal Notations (IF)	✓	✓	✓	✓		
	Formal Notations (F)			✓	✓	✓	✓
	Transition from IF to F					✓	✓
	Version Management			✓		✓	
	Multi-User Interaction	✓	✓	✓	✓		
	Remote Interaction	✓	✓	✓	✓		
	Saving Design as XMI					✓	✓
	Sharing Design			✓	✓		

In this paper, we address the following research question:

- RQ2. Can supporting the mix of informal and formal modeling notations make the modeling tools of MBSE easier to use and more productive?
  - RQ2.1. Do developers find combining informal and formal modeling beneficial?
  - RQ2.2. How do developers perceive the usability of the proposed solution, *OctoUML*?

We conduct User Studies [81] to evaluate *OctoUML*. *OctoUML* is perceived to have the potential to effectively support the activities of software developers by supporting the creation and mixing of both informal and formal modeling notations. Moreover, the perceptions on *OctoUML* ' multiple modes of interaction are positive. We assess the usability of *OctoUML* using the System Usability Scale (SUS) [86]. The results show that *OctoUML* provides a usable and user-friendly environment. Chapter 3 provides more details on the characteristics and functionalities of *OctoUML*, as well as on the evaluation process and the obtained results.

1.6.3 Paper C: Model-Based Software Engineering: A Multiple-Case Study on Challenges and Development Efforts

On the one hand, MBSE has been applied effectively in several application sectors, e.g., embedded systems [4] and telecommunication [23]. Furthermore, by focusing on practitioners' experiences and perceptions, several studies claimed that the adoption of MBSE helps to (i) improve the productivity of the developing teams by increasing the abstraction level, (ii) enhance the quality of the software, and (iii) support software maintainability [4, 21, 87]. On the other hand, some practitioners consider MBSE as a time-consuming and unproven approach that merely complicates matters [3].

Devanbu et al. [6] suggested that more in-depth studies that address the interplay of belief and evidence in software practices are needed. In this paper, we extend the goals of Paper A [85] in contributing to the body of knowledge on understanding the process and use of MBSE in practice. In particular, we conduct a multiple-case study [78] by collecting, analyzing, and discussing empirical data about MBSE efforts and challenges collected from 2 two-month MBSE projects. The main contributions of this paper are two-fold:

- Firstly, we shed light on the distribution of development efforts in MBSE. The resulting observations on effort distribution could lead to improved MBSE project planning and organization (e.g., resource allocation and risk management), which in turn could lead to cost reduction.
- Secondly, we report and further analyze different challenges to the process and use of MSBE in practice. Exposing such challenges would make them a candidate subject for research that is concerned with MSBE process improvement. Moreover, understanding and providing ways to overcome these challenges could bring a significant impact to the effectiveness and efficiency of MBSE.

In particular, we address the following research questions:

- RQ3. How is the total effort spent on MBSE distributed over different development activities?
  - RQ3.1. How is the effort spent on different MBSE development activities distributed over time?
  - RQ3.2. How large is the portion of collaborative work in MBSE projects?
  - RQ3.3. What are the challenges that affect MSBE in practice?

We show that there is no effort penalty in building models as part of the construction phase. We also show that the majority of MBSE effort is spent

on communication between developers on MBSE activities, such as project management, design, coding, testing, and configuration and integration. The resulting observations on effort distribution of this study could lead to improved MBSE project planning and organization, which in turn could lead to cost reduction. Our inquiry into challenges showed that tool-related challenges are the most encountered. We uncover that specific tool-challenges are due to:

- usability of the tools,
- the learning of the tool-chain,
- the interoperability of various tools, and
- the installation and configuration of the tools.

Based on these findings, we study – in Paper D [88] – how to improve the usability of MBSE tools, as well as investigate new approaches to reduce the effort of MBSE activities.

1.6.4 Paper D: OctoBubbles: A Multi-view Interactive Environment for Concurrent Visualization and Synchronization of UML Models and Code

Developers navigate both code and design. Most of the software modeling tools like ArgoUML[3], Visual Paradigm[4] and ObjectAid[5] support forward, reverse and round-trip engineering [89]. However, these tools do not provide developers with multiple simultaneously-visible views of both low- and high-level software artifacts (i.e., source code and design models). The developers who use these tools are constrained to use different applications in different windows making the workspace overcrowded with many opened interfaces and tabs. In such a situation, developers often lose the big picture and spend unnecessary effort on navigation and locating the artifact of interest.

In this study, we build on the tool that we developed in Paper B [83] to create *OctoBubbles*. *OctoBubbles* is an interactive environment for concurrent visualization and synchronization of both high- and low-level software artifacts. It aims to assist program comprehension and tackle the problem of software navigation.

The main contributions of this paper are two-fold:

- First, a design of a novel *scaling* approach which is used to provide an interactive, bidirectional, and smooth navigation between software models and code.

[3] `http://argouml.tigris.org/`
[4] `https://www.visual-paradigm.com/`
[5] `http://www.objectaid.com/`

- Second, the results of a qualitative user evaluation of *OctoBubbles* indicating a high level of interest, and pointing out to potential benefits and future improvements.

In particular, we address the following research questions:

- RQ2. Can supporting a smooth navigation between models and code make the modeling tools of MBSE easier to use and more productive?
  - RQ2.1. Do developers find navigating between models and code beneficial?
  - RQ2.2. How do developers perceive the usability of the proposed solution, *OctoBubbles*?

Figure 1.11 shows the main interface of *OctoBubbles*. It provides an overview on how the UML model and the corresponding source code are concurrently visualized on the same canvas. Chapter 5 provides more details on the design and functionalities of *OctoBubbles*.

The participants who evaluated *OctoBubbles* stated that it helps to better establish traceability between a model and its associated source code, especially

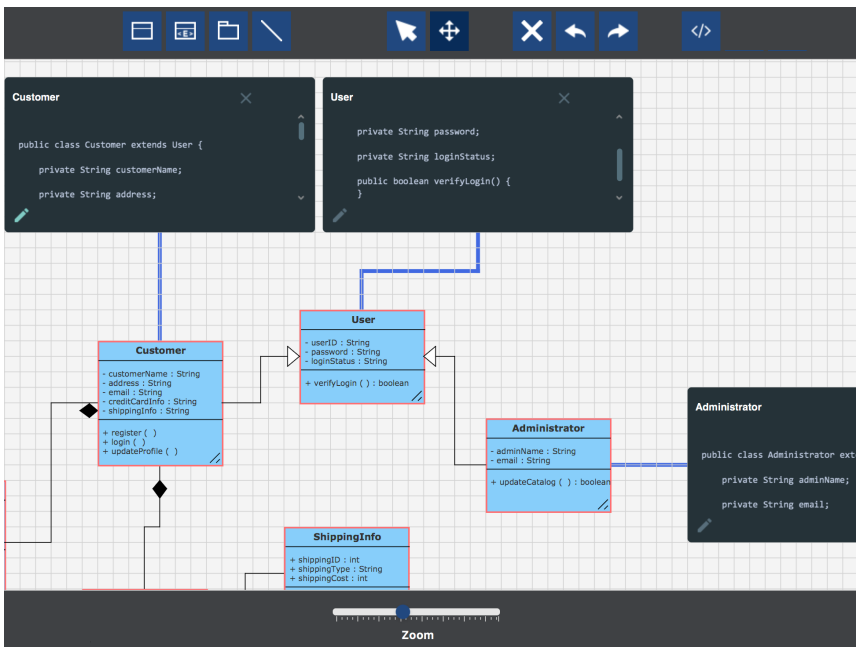


Figure 1.11: A part of the main canvas of OctoBubbles. The buttons at the top are (*from left to right*): create class, create enumeration class, create package, create association, select, pan, delete, undo, redo and visualize code.

in software testing and maintenance tasks. For such tasks, developers might jump back and forth between low- and high-level artifacts to have a better understanding of the system. Moreover, the visualization approach of *Octo-Bubbles* was considered extremely useful to have an abstract view of a software system, concurrently with a more detailed view of its aspects. *OctoBubbles* was perceived more efficient than other CASE tools that support forward, reverse and round-trip engineering. Also, the visualization mechanism of *OctoBubbles* was considered very helpful in saving the effort of artifacts navigation.

1.6.5 Paper E: Does Distance Still Matter? Revisiting Collaborative Distributed Software Design

Companies engage in global software engineering (GSE) to reduce development time and costs. Companies also head toward cross-site distribution of their development work to take advantage of proximity to markets and customers [25]. However, working at a distance might compromise the effectiveness of GSE [24].

Many researchers have explored the impact of distance on collaborative work. Herbsleb [24] argued that colocation fosters communication because developers are aware of who is around and who is doing what. In contrast, being unable to share resources and see what is happening at the other sites hinders communication across different locations.

As globally-distributed projects are becoming the norm in SE [40] and lead to social, technical, and organizational challenges [90], the software design activities are affected as well. Moreover, we are witnessing significant advances in communication and collaboration technologies. So, in this study we explore the design activities of both co-located and distributed professional software developers to understand:

- to what extent the design activities are hampered by the distribution of collaborating teams, and
- whether advances in collaboration and communication technology enable effective and efficient remote collaboration.

This study extends the goals of Paper C [27] in contributing to the body of knowledge on understanding software design in MBSE by analyzing collaborative, distributed software design. Moreover, we use the tool that we developed in Paper B [83] and extended in Paper D [88] in the investigation of the activities of collaborative, distributed design.

In this study, we address the following research questions:

- RQ4. How does distance influence the design activities of software developers?
  - RQ4.1. How does distance influence the amount of problem- and solution design decisions?

- RQ4.2. How does distance influence design communication?
- RQ4.3. What challenges are encountered when collaboratively designing software at a distance?

To answer these questions, we conducted a multiple-case study exploring in depth the design process of co-located and distributed software developers in a collaborative design setting. For the first case, we used the data set provided by [59], who performed the study with three (co-located) teams of two professional software developers each. For the second case, we recruited three teams of two professional software developers to work on the same design challenge, but from two different geographic locations: Aachen, Germany and Gothenburg, Sweden. Instead of a regular whiteboard, we used interactive whiteboards with a simplified version of *OctoUML*. While *OctoUML* has some UML capabilities, like creating class shapes, we removed those in the study to make *OctoUML* resemble a regular whiteboard as closely as possible. Immediately after each distributed design session, we asked the developers to evaluate the usability of OctoUML. The reason for this is to understand to what extent the usability of OctoUML affected the work of the distributed developers. In particular, we asked the developers to answer the System Usability Scale (SUS) questionnaire. Overall, we observe that the perceptions regarding the usability of OctoUML were positive. Regarding the SUS score, OctoUML received an average SUS score of 74.17 ± 5.63, which can be interpreted as a grade of $B-$ (i.e., a good usability score) according to [86].

Our findings indicate that co-located developers discuss more design decisions in the problem domain than distributed developers. Moreover, co-located teams have many *creative conflict* discussions which promote software design reasoning and enhance the effectiveness of group tasks. Accordingly, we suggest that geographic distribution of collaborating partners in practice still raises social and technological challenges. We argue that the social challenges in distributed design might be related to lack of trust and lack of common understanding. Moreover, we observed that lack of awareness was the main technological challenge in the distributed setting. Thus, we suggest that modern collaborative design environments should focus on supporting awareness, such as the ability for developers to relate to each other through pointing, gaze, and gestures. In this direction, Trainer and Redmiles [91] propose a tool prototype, that is focused on two kinds of information: availability and responsiveness of collaborators. The goal of the tool is to bridge the gap between awareness and trust in globally distributed software teams.

Moreover, by observing the distributed design sessions we noticed that the distributed developers use informal notations for problem domain exploration and formal notations (i.e., geometric UML shapes) for design solution representation. Thus we underline that software design and modeling tools should enable the creation of both informal and formal notations to support problem– and solution space exploration.

Chapter 6 provides more details on our study of the collaborative design activities of co-located and distributed software developers.

1.6.6 Paper F: Software Engineering Whispers: The Effect of Textual Vs. Graphical Software Design Descriptions on Software Design Communication

Software engineering is a social activity and requires intensive communication and collaboration between developers. In large companies, developers work in different development teams and collaboratively communicate with many stakeholders. In such a setting, the quality of communication between the stakeholders plays an important role in reducing the overall effort of teams and development projects. Furthermore, poorly defined software applications (due to miscommunication between stakeholders) can affect the final structure and/or behavior of these applications. This is in line with Jarboe et al. [92] and Kortum et al. [93] who consider that the quality of communication does influence developers' activity and achievement, and therefore customer's satisfaction. The effort spent on discussions and communication, as we found in Paper C [27], is actually more than all of the efforts that developers spent in any of the other observed development activities. The aforementioned studies underline the importance of communication in Software Engineering. They also highlight the need to study communication in-depth to determine elements or criteria of its efficiency and effectiveness.

In MBSE, the software design knowledge is often communicated by means of graphical software design descriptions (i.e., software models). In other software engineering approaches which are not model-based, the software design knowledge is often communicated by textual software design descriptions. Graphical descriptions encode and present knowledge differently from textual descriptions. Moreover, these two types of knowledge representation are differently processed by the human mind, as stated by Moody [28]. Empirical evidence on how graphical descriptions affect developer's achievement and development productivity is still underwhelming, as reported by Hutchinson et al. [5]. Moreover, Melià et al. [29] report that the software engineering field lacks a body of empirical knowledge on how different representations (graphical vs. textual) could provide support for improving software quality and development productivity.

In this paper, we extend the goals of Paper C [27] in contributing to the body of knowledge on understanding the design activities of MBSE in practice. In particular, we investigate how different software architecture design descriptions (graphical vs. textual) influence the communication of design knowledge.

With respect to knowledge communication, we look into the following *communication* aspects:

(a) *Explaining*: or knowledge donating, communicating the personal intellectual capital from one person to others [94].

(b) *Understanding*: or knowledge collecting, receiving others' intellectual capital [94].

(c) *Recall*: or memory recall, recognizing or recalling knowledge from memory to produce or retrieve previously learned information [95].

(d) *Collaborative Interpersonal Communication* [96], which includes:

- *Active Discussion*: questioning, informing, and motivating others.
- *Creative Conflict*: arguing and reasoning about others' discussions.
- *Conversation Management*: coordinating and acknowledging communicated information.

In this study, we address the following research question:

- RQ5. How does the representation of software design (graphical vs. textual) influence design communication?

Based on empirical findings, we suggest that using a graphical software design description is better for:

- promoting *Active Discussion* between developers,
- reducing *Conversation Management* effort, and
- improving the *Recall*-ability of design details.

Furthermore, we found that *motivating* textual design descriptions (by adding design rationale) and making them *cohesive* (by organizing the design knowledge in the document) helps to enhance the explaining and recall of their details.

Finally, we observe a difference in the explaining approach of software design knowledge represented by a graphical vs. textual description. Figure 1.12 provides an illustration of the observed explaining approaches of a textual (TSD) vs. graphical (GSD) Model-View Controller (MVC) design. In one approach, the *Explainers* of a TSD tended to explain the three modules of the MVC sequentially: Firstly the *Model* entities, then the *Controllers*, and lastly the *Views*, as these modules are orderly presented in the textual document. We think that this trend is intrinsically imposed by the nature of textual descriptions where the knowledge is presented sequentially on a number of consecutive ordered pages. In the other approach, the *Explainers* of the GSD had more freedom in explaining the design. Indeed according to their explaining preferences, the *Explainers* of the GSD tended to jump back and forth between the three MVC modules when explaining the design. Based on this, we suggest that a GSD has an advantage over the TSD in unleashing *Explainers*' expressiveness when explaining the design, as well as in helping navigation and getting a better overview of the design.

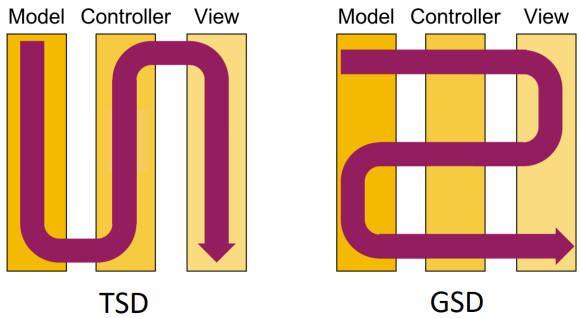


Figure 1.12: Observed explaining approaches of TSD vs. GSD

1.7 Conclusion and Future Work

The goals of this PhD research are to *understand* and *support* software design in model-based software engineering practices. Based on the results of this PhD research, we present a number of findings and suggestions to MBSE researchers and practitioners. These findings and suggestions are related to *MBSE process*, *MBSE artifacts*, and *MBSE tools*.

(A) MBSE process:

- We find that a significant part of creating models is devoted to design thinking about the problem domain and solution domain. Accordingly, modeling should not be considered as a costly process, because it triggers design thinking. It is an open question how much design thinking happens when modeling is not used. But, when modeling is not used then the code is only produced by the coding process. Thus, we suggest comparing or studying the design thinking when modeling vs. coding.
- We find that the MBSE approach does not require a lot of effort on design and modeling. Moreover, this approach requires only little effort on manual coding, as most of the code is obtained from models via code-generation.
- We find that model-based software development is an endeavor that requires intensive communication and collaboration between developers. Therefore, we suggest studying mechanisms for improving the efficiency and effectiveness of communication which in turn would improve the quality and increase the productivity of the overall software engineering process.
- We suggest practitioners engaging in collaborative design to be aware that geographic distance can hamper the design process. To prevent

this, organizations might decide to take important architectural decisions in co-located settings only.

- In addition to the geographic distance, cultural and social barriers can hamper design collaboration. Therefore, we recommend organizations to:
  - Consider cultural training for their developers.
  - Establish trust via arranging personal, virtual meetings, or social events before the remote design sessions.
  - Establish common ground via exchanging interests, experiences, expertise, and beliefs between distributed designers.
- We also suggest researchers to study, in depth, the social challenges in MBSE, and investigate approaches to account for the effect of these challenges on team behavior and development activities.

(B) MBSE artifacts:

- We find that using graphical software design descriptions (e.g., software models) to communicate software designs produces more active discussion, less conversation management, and better recall.
- We suggest the use of graphical software design descriptions in software design meetings in order to enhance communication and, therefore, increase the productivity of software development teams.
- We encourage developers to include design rationale in design documentations to improve design communication, which in turn should improve the overall communication and collaboration, and thus the productivity, in SE projects.
- We find that a graphical software design description has an advantage over the textual software design description in unleashing the expressiveness of design explainers when explaining the design, as well as in helping navigation and getting a better overview of the design.

(C) MBSE tools:

- We suggest to reduce the complexity of software design tool-support and enhance the usability thereof. This would let developers spend more time on pondering and thinking of design decisions.
- The integration of informal and formal modeling in OctoUML is perceived beneficial by developers for supporting the design process and its flow. Thus, we suggest enabling modeling tools to support both informal and formal modeling.

- We find that tool-related challenges are the most encountered challenges in MBSE. These tool-challenges are due to: tools usability, tool-chain learning, interoperability of tools, and tools installation and configuration. Providing approaches to overcome these challenges could bring a significant impact to the effectiveness and efficiency of the MBSE approach.
- The navigation mechanism between software models and code of OctoBubbles is perceived beneficial by developers for supporting software understanding and facilitating traceability. Thus, we suggest enabling MBSE tools to support navigation between different software engineering artifacts.
- To enhance the effectiveness of distributed software design, we recommend the developers of collaborative MBSE tools to support awareness by adapting technology to provide immersive telepresence experiences.
- We suggest the introduction of explicit triggers for effective communication in co-located and distributed software design to enhance the process of making design decisions.

1.7.1 Future Work

In this section, we describe a number of future research directions that can extend the work of this thesis. Figure 1.13 is an extension of Figure 1.8 that we illustrated previously in Section 1.6 (Contributions). It shows three future work (FW) directions to understand and support the design activities of software developers in MBSE. These directions mainly extend Paper E [97] and Paper F [98].

- **FW 1**: Based on the findings of Paper E ("*Does Distance Still Matter? Revisiting Collaborative Distributed Software Design*"), we started investigating the use of Virtual Reality (VR) technology to assist collaborative, distributed software design. In particular, we are collaborating with researchers from the Slovak University of Technology on evaluating a VR-prototype of OctoUML. The main goal of this research direction is to provide an immersive telepresence experience and support awareness in remote design collaboration.

- **FW 2**: Based on the findings of Paper E, we argue that, despite the technological advances in collaborative MBSE, effective collaboration can only be achieved if we understand how to account for social barriers. Thus, we propose to study, in depth, how these barriers affect the design activities, and how their effects can be reduced.

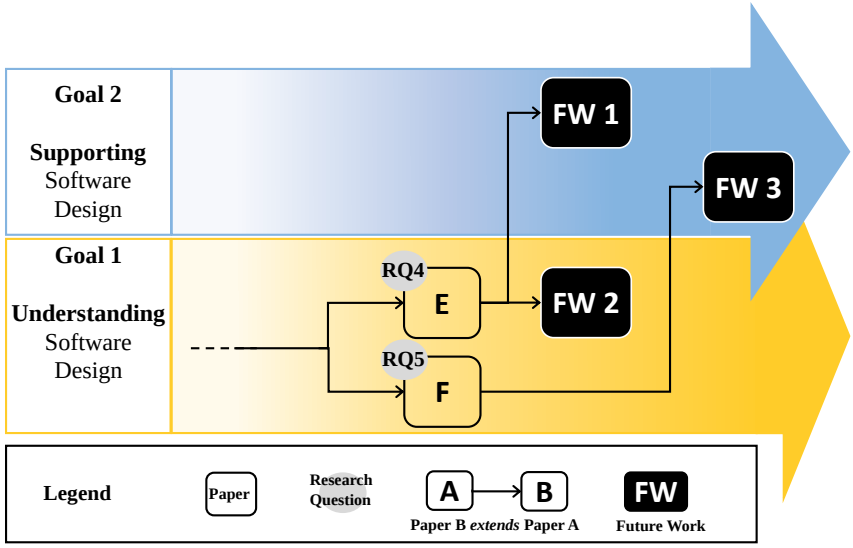


Figure 1.13: Future research directions.

- **FW 3**: Based on the findings of Paper F ("*Software Engineering Whispers: The Effect of Textual Vs. Graphical Software Design Descriptions on Software Design Communication*"), we suggest investigating new techniques or approaches for supporting software design communication. One example of these approaches is proposed in a study by Tang et al. [99] where a reminder card approach was employed to improve software design reasoning discussions. Another example is proposed by Robillard et al. [100] who argue that automatic on-demand documentation generators would effectively support the information needs of developers.

Chapter 2

Dissecting Design Effort and Drawing Effort in UML Modeling

(A) R. Jolak, E. Umuhoza, T. Ho-Quang, M.R.V. Chaudron, M. Brambilla "Dissecting Design Effort and Drawing Effort in UML Modeling" *In the 43th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 384-391. 2017.*

Abstract

One argument in the discussion about the adoption of UML in industry is the supposedly large effort it takes to do modeling. Our study explores how the creation of UML models can be understood to consist of different cognitive activities: (i) *designing*: thinking about the design (ideation, key-design decision making), (ii) *notation expression*: expressing a design in a modeling notation and (iii) *layouting*: the spatial organization of model elements in a diagram. We explain that these different subactivities relate to different short-term and long-term benefits of modeling. In this study we present two controlled experiments with a total of 100 subjects creating models for a small system. In these experiments we focus on software models as represented through UML class diagram. Our results show that *at least* 56% of the effort spent on creating a class model is actually due to *designing*. *Notation expression* is around 41% of the model creation effort and *layouting* is in the order of 3%. This finding suggests that a significant part of creating models is devoted to design thinking about the problem.

Keywords: Software Engineering; Software Modeling; Software Design; Modeling Effort; Designing Effort; Design Thinking; UML

2.1 Introduction

Models have emerged in software engineering as a powerful tool to tackle complexity of system specifications. Indeed, modeling allows to address the description of software based on different levels of abstraction and from multiple perspectives, in order to accommodate the needs of communication and description of a variety of stakeholders. In fact, models help to describe, reason, predict and evaluate both software problems and solutions. Furthermore, they provide effective means for supporting the communication between stakeholders, and serve as specifications for implementation [3]. However, software practitioners consider such approaches *time-consuming*, hence prefer to avoid using models which are believed as complex, inconsistent, excessive and unnecessary artifacts [101].

In this study we want to find out how much of the modeling effort is spent on the *design* of the solution (i.e. pondering and making the design decisions). If a significant part of the creation of the models is devoted to design thinking about the problem, it could mean that the fault of supposedly unproductive processes should not be blamed on modeling, but to the (anyhow necessary) effort devoted to thinking about the problem and identifying the solution (i.e. *design* effort).

In order to assess this, we run a set of experiments about the creation of models in response to a set of requirements: we measure the effort required to make the initial model of a system (*modeling* effort), and then we measure the effort required to recreate the same model again, simply by redrawing the already defined solution (*copying* effort). For the *copying* of the solution, we assume a subject does not have to (re)do any design thinking, but only spend effort on entering a solution into a modeling tool. At the end we calculate *design* effort by assessing the time difference between the two activities (see the details in Section 2.3).

Some empirical studies provided evidence on the benefits of UML modeling in enhancing the productivity, quality and maintenance of software products [5, 102–104]. These studies sustain that benefits of UML modeling take place after a *long-term*, in the sense that UML modeling introduces an initial overhead at the beginning, whereas the benefits start to take place at late stages – like the *two* marshmallows reward of the *'one marshmallow now or two marshmallows later'* experiment [105]. The minority of the kids who participated in that experiment preferred to have one marshmallow now rather than two later. Such behaviour may be similar to that of software developers' who do not prefer to spend time on modeling at early stages, and eat the marshmallow immediately.

Our hypothesis is that the benefits of UML modeling does not only take place after a long-term, but also immediately at early stages. To assess this hypothesis, we try to dissect *design*, *notation expression* and *layout* efforts in UML modeling. If *design* effort dominates the modeling process, then UML modeling

consists mostly of thinking of the domain problem and identifying/designing the solution. In other words UML modeling would reward *three* marshmallows; one immediately (foster design thinking and promote ideation) and two later (enhance productivity, quality and maintenance).

This paper is organized as follows: in Section 2.2 we discuss the related work. We describe our approach in Section 2.3. Section 2.4 illustrates the design of our experiments and details their operational phases. Section 2.5 reports the results of the experiments, which are then discussed in Section 2.6. We consider the threats to validity in Section 2.7. Finally, we conclude and discuss the future work in Section 2.8.

2.2 Related Work

More often than not, evaluation of software modeling practices and associated effort have been left in the realm of myth. As a result, software developers and also modeling experts have different opinions on the pros and cons of modeling that rely on beliefs (i.e. factoids) more than facts. This leads to a variety of situations where no proper guidance can be provided in the selection of the appropriate design tools for software. More in general, the whole field of software engineering perceives the discrepancy between scientifically validated results (e.g., in the empirical software engineering) and developers beliefs, usually based only on personal perspectives on the development processes. Various recent studies demonstrate that more in-depth studies that address the interplay of belief and evidence in software practices are needed [6].

A few studies addressed the monitoring and analysis of modeling practices. Sharif et al. [61] explored design strategies and types of activities that designers engage in during software design sessions. They used video-recordings and transcripts of three two-person teams who were assigned to create a software design for the same set of requirements on a whiteboard. In addition to the identified design activities, they also found the sequence of activities for each session as well as the activity that took the longest by analyzing the duration of actions and speeches mapped to various design activities. They identified some of time-consuming activities such as decisions about the logic, discussion of uses cases, drawing class diagrams, and drawing the user interface.

Some experiments have been conducted to identify strategies during the modeling task. The works [106,107] recorded the activities of a pool of students when creating UML class diagrams and analysed the logs using LogViz. Four different strategies were found, namely Depth First, Breath First, Depthless and Adhoc. The study also found that students spent most of their time in understanding assignment tasks and in defining the layout of the model. One thing that the log failed to say was what students do in the time gaps where they appeared to do “no activity”.

Despite the resistance of software companies in adopting the model-driven

development, few industrial success stories can be found. Brambilla and Fraternali [108] presented the industrial success stories (spanning from financial and banking to utility) and the advantages of adopting Model-Driven Engineering (MDE) perceived by the customers of WebRatio, a company which focuses on MDE tools and services since 2001. The study also included a report of amount of the effort dedicated by the designers to the different modeling activities.

Furthermore, few researches evaluated some of the claimed advantages brought by MDE. Diaz et al. [109] measured the reuse gains brought by MDE in comparison with manual coding of blogs. Brambilla et al. [110] analyzed the productivity gain brought by MDE in comparison of manual coding of cross-platform mobile applications. They observed that for mobile applications, model-driven development allows to save more than 20% of the cost.

2.3 Approach

In this section, we provide an overview of the approach used to estimate the effort devoted to different activities in the process of software modeling. Our approach considers software modeling as a process that encompasses three different activities:

(A) Designing of the solution: It represents the activity of reasoning and thinking about a design solution of a domain problem. We call the time devoted to this activity: *Design Effort (DE)*.

(B) Notation expression: The expression of the identified solution through a modeling notation. We call the time devoted to this activity: *Notation Expression Effort (NEE)*.

(C) Layouting: It represents the activity of organization of the model elements in a diagram (e.g. to enhance the readability of the model). We call the time devoted to this activity: *Layout Effort (LE)*.

Based on that, the total effort dedicated to the software modeling process is simply obtained as a sum of the single efforts spent in each modeling activity. The total Modeling Effort *(ME)* in given by Equation Eq.A.

$$ME = DE + NEE + LE \tag{Eq.A}$$

To compute the effort spent in each activity, we ran two-phase experiments. In the first phase, we measure the effort required to make the initial model of a system (*modeling* effort). While in the second phase, we measure the effort required to recreate the same model again, simply by redrawing the already defined solution (*copying* effort). At the end, we calculate the *design*, *notation expression*, *layout* efforts by assessing the time difference between the two phases.

2.3.1 Phase 1: Modeling

During this phase the participants are asked to create a UML class diagram that addresses a simple assignment using a modeling tool. The participants think about the solution, express their solution through a modeling notation, and may organize the elements of the model on the canvas.

Let us denominate the set of all persistent elements that are part of the final model with (Π), and the set of all deleted elements that are not in the final model with (Δ). Let us also denominate the set of all elements (persistent and deleted) with (Σ). We have that:

$$\Sigma = \Pi \cup \Delta \tag{Eq.B}$$

Based on Equation Eq.A and Equation Eq.B, the effort dedicated to the modeling phase, called *modeling effort (ME)*, is given by the following equation *(unknowns are in bold, whereas the known efforts are obtained via analyzing the log of the modeling tool)*:

$$\text{ME}(\Sigma) = \mathbf{DE}(\Sigma) + \mathbf{NEE}_m(\Pi) + \mathbf{NEE}_m(\Delta) + LE_m(\Pi) + LE_m(\Delta) \tag{Eq.1}$$

Where:

- ME(Σ): *(known)* the total modeling effort,
- DE(Σ): *(unknown)* design effort, the time spent on thinking about the solution (including both persisted and deleted elements),
- NEE$_m$(Π): *(unknown)* notation expression effort of persistent elements during modeling phase; the time spent on creating elements that are part of the final model,
- NEE$_m$(Δ): *(unknown)* notation expression of deleted elements during modeling phase; the time spent on creating elements that are not in the final model (deleted because of exploring design alternatives),
- LE$_m$(Π): *(known)* layout effort of persistent elements during modeling phase; the time spent on organizing elements of the final model,
- LE$_m$(Δ): *(known)* layout effort of deleted elements during modeling phase; the time spent on organizing elements that are not in the final model.

2.3.2 Phase 2: Copying

During the copying phase, the participants are asked to simply re-draw (copy) the same modeling solution produced in phase 1. In this phase the participants are asked to do a strict copy without thinking or enhancing the identified

solution in phase 1. Thus, the effort dedicated to this phase, called *copying effort (CE)*, is obtained via the following equation:

$$CE(\Pi) = \mathbf{NEE}_c(\Pi) + LE_c(\Pi) \quad \text{(Eq.2)}$$

Where:

- $\mathrm{CE}_c(\Pi)$: *(known)* the total copying effort,
- $\mathrm{NEE}_c(\Pi)$: *(unknown)* notation expression effort of persistent elements during copying phase,
- $\mathrm{LE}_c(\Pi)$: *(known)* layout effort of persistent elements during copying phase.

2.3.3 Analyze Effort Difference

By isolating *design* and *layout* efforts, the *notation expression* effort of persistent elements is the same in both phase 1 and 2. We have that $\mathrm{NEE}_m(\Pi) = \mathrm{NEE}_c(\Pi)$. We compute the *design* effort by subtracting Equation Eq.2 from Equation Eq.1. The final result is reported as follows:

$$\mathbf{DE}(\Sigma) = ME(\Sigma) - CE(\Pi) - LE_m(\Pi) - LE_m(\Delta) - \mathbf{NEE}_m(\Delta) + LE_c(\Pi) \quad \text{(Eq.3)}$$

First of all, $\mathrm{LE}_m(\Pi) + \mathrm{LE}_m(\Delta) = \mathrm{LE}(\Sigma)$ is the total layout effort (LE) in the modeling phase. Now let us consider the number of persistent and deleted elements in the modeling phase as $|\Pi|$ and $|\Delta|$, respectively. Based on Equation Eq.2, we identify $\mathrm{NEE}_m(\Delta)$ via the following equation:

$$\mathbf{NEE}_m(\Delta) = \frac{|\Delta|}{|\Pi|} \cdot \mathrm{NEE}_c(\Pi) \quad \text{(Eq.4)}$$

The (DE) is given by inserting the value of $\mathrm{NEE}_m(\Delta)$ in Equation Eq.3. Furthermore, considering Equation Eq.A, the notation expression effort is given by the following equation:

$$NEE = ME - DE - LE \quad \text{(Eq.5)}$$

We are interested in identifying how much of the total modeling effort is spent on *designing*, *notation expression* and *layouting*. Thus, we can define the Design Effort Percentage (DEP) as the ratio of the Design Effort (DE) over the total Modeling Effort (ME):

$$DEP = DE/ME \quad \text{(Eq.6)}$$

Similarly for NEE and LE, the percentages are given by:

$$NEEP = NEE/ME \quad \text{(Eq.7)}$$

$$LEP = LE/ME \quad \text{(Eq.8)}$$

At this point, we want to underline that the DE may also occur during the process of UML notation expression and/or layouting (as we are capable of thinking while drawing). Our calculations indeed estimates the lower bound on DEP (the *minimum* DEP), in the sense that we do not assume any occurrence of DE during the process of notation expression and/or layouting. So the real value of DEP may be more than the minimum found. At maximum, DE could occur continuously from the beginning to the end of the modeling process (i.e. Max(DEP) is 100%).

Our experiments were conducted at Polytechnic University of Milan in Italy and Gadjah Mada University in Indonesia. We formulated three different modeling scenarios. Every scenario describes a system to be designed (see Section 2.4.1.1). A mix of 100 B.Sc. and M.Sc. software engineering students were randomly given the modeling scenarios. To create their models, students were asked to use WebUML [106] and Papyrus (https://eclipse.org/papyrus) modeling tools. Both tools allow the logging of the modeling activities. We collected the recorded log files for each participant and assignment. We also setup an online questionnaire through which participants answered questions about their background, expertise in UML modeling, tool usability, and assignments understandability.

Based on the collected results, our research objective has been addressed by defining and responding to the following research questions:

1. How much of the modeling effort is *design*, DEP?
2. How much of the modeling effort is *notation expression*, NEEP?,
3. How much of the modeling effort is *layout*, LEP?
4. Does the *size* of the modeling scenario affect DEP, NEEP and LEP?
5. Does the *topic* of the modeling scenario affect DEP, NEEP and LEP?

2.4 Experiment

This section describes the modeling experiments conducted to answer the research questions presented in Section 2.3. For this study, we conducted the experiments in two different settings: (i) participants create models *Individually* and (ii) in teams *Collaboratively*. We refer to the former setting as *EXP1* while the latter as *EXP2*.

EXP1 was conducted at Polytechnic University of Milan involving 48 students. During this experiment, the participants were asked to design a

solution for given modeling scenarios using the *WebUML editor*. EXP2 was conducted in Gadjah Mada university in Indonesia involving 13 groups of 4 students each. Each group was asked to design a solution for a given modeling scenario using Papyrus.

2.4.1 Experiment Preparation

2.4.1.1 Scenarios Definition

To make our analysis of the specific case independent, we evaluated the *design* effort with three scenarios (*scenario 1*, *scenario 2* and *scenario 3*) from different topics and with slightly different size (the number of classes in their solution is different by one or two). Every scenario describes a simple system to be designed. In addition, we have defined a *test scenario*, used at the beginning of the modeling sessions to explain to the participants how the tool works and to let them get familiar with it. The description of the scenarios and the experimental material can be found here: (`https://goo.gl/mvz2bm`).

2.4.1.2 Assigning scenarios to participants

EXP1 The ideal strategy to get the most generalizable results is to assign all the three scenarios to each participant. However considering the average time required to complete one scenario (around 25 minutes), assigning three scenarios to each participant was not feasible due to the limited time the students had available for the experiment. Thus, we decided to assign two scenarios to each participant. In order to limit unintended effects and to have balanced experiments, we used the Graeco-Latin square theory [111] by assigning different orders of scenario's to different groups of students. The scenarios that are used in this experiment are: scenario 1, scenario 2 and scenario 3.

EXP2 The purpose of this setting is to study possible effects of group work and modeling tools on the software modeling effort. The used scenario in this experiment is: scenario 2. The obtained data from this experiment are compared to the data that are related to scenario 2 of EXP1.

2.4.2 Experiment Execution

In this phase, the participants model the assigned scenarios using the WebUML tool for *EXP1* and Papyrus for *EXP2*. Both tools have a logging feature that logs the participants' actions (such as the creation, modification, and deletion of an element). The logs are useful to derive quantitative data which enable us to compare and evaluate the produced designs and the time spent on interacting with the tool. The experiments were conducted following these five steps:

(A) *Introduction.* We introduced the modeling tool to the participants through a training session.

(B) *Instruction.* During this phase we explained the procedure of the experiment to the participants, as well as showed them how to save and submit their designs. Then, a short training exercize of 15 minutes took place under our supervision in order to get the participants accustomed with the basic functionality of the tool. This was done using a test scenario, equal for all the participants.

(C) *Modeling assigned scenarios* (See Section 2.3, *Modeling phase*). The participants have to model the assigned scenario/s.

(D) *Copying assigned scenarios* (See Section 2.3, *Copying phase*). The participants simply copy (re-draw) the proposed model solution that is produced in the *Modeling Phase.*

At the end of step 4, the participants of *EXP1* were asked to proceed with the *modeling* and *copying* of the other scenarios following the same way as described in steps 3 and 4, respectively.

(E) *Closure.* after submitting their models, the participants were asked to answer a questionnaire about their personal information, knowledge about UML modeling and perception regarding the usability of the modeling tools (WebUML for *EXP1* and Papyrus for *EXP2*).

The experiments were performed in a controlled environment. The participants worked on computers in a lab at both Universities. There were supervisors that walked around to monitor that the participants worked on the assignment and not on other tasks or distractions.

2.5 Results

In this section we report the results of the two conducted experiments (EXP1 and EXP2). For EXP1, we present the results of 37 subjects since the experiments of 11 subjects were not valid (8 worked concurrently on the first and second task while 3 had technical network problems which prevented us from receiving their data) and then removed from the data set. We used the statistical package R [112] to perform all tests. We chose a significance level at 0.05, which corresponds to a 95% confidence level.

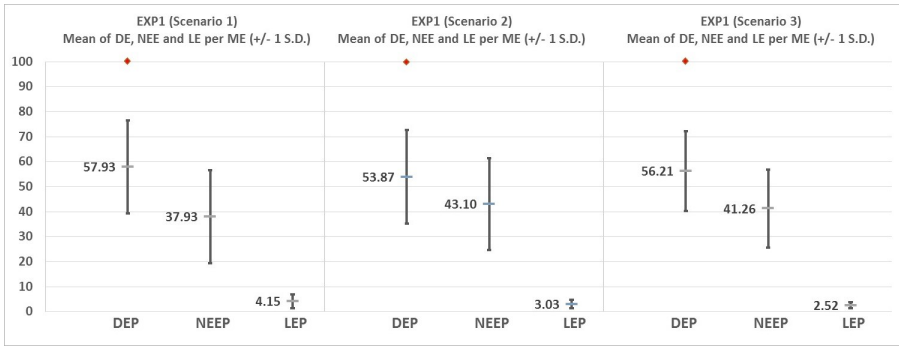


Figure 2.1: Efforts distribution related to the scenarios of EXP1

Table 2.1: Statistical results for all scenarios of EXP1 and EXP2

EXP1							
Scenario	**# Subjects**	**DEP**		**NEEP**		**LP**	
		Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
Scenario 1	21	57.93	18.66	37.93	18.65	4.15	2.69
Scenario 2	25	53.87	18.72	43.10	18.39	3.03	1.75
Scenario 3	22	56.21	16.03	41.26	15.67	2.52	1.20
All		55.88	17.69	40.91	17.51	3.21	2.04
EXP2							
Scenario	**# Subjects**	**DEP**		**NEEP**		**LP**	
		Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
Scenario 2	52	73.60	8.40	22.14	8.34	4.25	2.88

2.5.1 Design, Notation Expression and Layout Efforts

2.5.1.1 EXP1

For each modeling scenario used during EXP1, we calculated the mean and the standard deviation of DEP, NEEP and LEP. The results are presented in Table 2.1.

2.5.1.2 EXP2

We calculated the mean and the standard deviation of DEP, NEEP and LP that are related to scenario 2 used in experiment EXP2. The results are presented in Table 2.1. Figure 2.1 and 2.2 provide a better view of the distributions of the various efforts (DEP, NEEP and LEP) related to the scenarios used in EXP1 and EXP2, respectively. (Note that the *red diamond* represents the Max(DEP), as the DE may occur concurrently with notation expression and/or layouting

activities.)

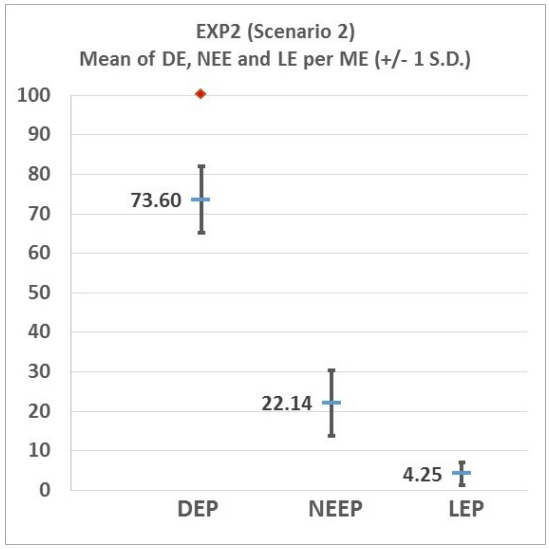


Figure 2.2: Efforts distribution related to scenario 2 of EXP2

Results for Q1, Q2 and Q3 (EXP1): For all scenarios, we found that the average *Design Effort* is 55.88%. While the rest is: 40.91% *Notation Expression Effort* and 3.21% *Layout Effort.*

Results for Q1, Q2 and Q3 (EXP2): the average *Design Effort* for scenario 2 is 73.60%. While the rest is: 22.14% *Notation Expression Effort* and 4.25% *Layout Effort.*

2.5.1.3 Quality of the models

We consider the quality of the produced models as a further crucial factor. We wanted to know how well the models reflect the domain problem, because extremely bad (or rough) models could affect the effort statistics. To grade the quality of the models, we defined a rubric before running the experiment. The rubric consists of a 5-point scale grading guidelines (`https://goo.gl/mvz2bm`). In advance of the actual grading a set of possible ideal solutions was discussed. The grading was done in two steps: (i) the assessors graded all models separately (independently), (ii) the assessors discussed the differences in grading and gave the model the final mark. Cohen's κ [113] was run to determine if there was agreement between the assessors. See Table 2.2 (the range of ratings is [1 to 5] where 1 is the most negative score and 5 is the most positive score, ratings are

Table 2.2: Quality of the models

						I-R. R.	
EXP	**Scenario**	**M**	**Q1**	**Q3**	**I-Q. R.**	***Kappa***	**p-value**
1	Scenario 1	4.00	3.00	4.00	1.00	0.57	0.000
1	Scenario 2	3.00	3.00	4.00	1.00	0.39	0.000
1	Scenario 3	4.00	3.00	4.00	1.00	0.61	0.000
2	Scenario 2	4.00	3.00	4.00	1.00	0.53	0.002

reported in terms of: (EXP: Experiment, M: Median Score, Q1: 1st quartile, Q3: 3rd quartile, I-Q. R.: Inter-Quartile Range (Q3-Q1), I-R. R.: Inter-Rater Reliability).

2.5.2 Comparison between the results of EXP1 and EXP2

The same modeling scenario (*scenario 2*) was used in both experiments EXP1 and EXP2. However, there are two factors that could affect the DEP, NEEP and LEP: (α) the modeling tool and (β) number of involved subjects per modeling task. For EXP1, *WebUML* was used by individuals. Whereas for EXP2, *Papyrus* was used by thirteen groups (of 4 people each). In order to assess the effect of both factors (α and β) on the DEP, NEEP and LEP, we performed Mann-Whitney's non-parametric test as the data are not normally distributed (Shapiro-Wilk test' p-values are less than 0.05). The following hypotheses were formed:

- Null Hypothesis H_{01}: There is no statistically significant difference in the DEP of the two cases.
- Alternative Hypothesis H_{A1}: There is a statistically significant difference in the DEP of the two cases.
- Null Hypothesis H_{02}: There is no statistically significant difference in the NEEP of the two cases.
- Alternative Hypothesis H_{A2}: There is a statistically significant difference in the NEEP of the two cases.
- Null Hypothesis H_{03}: There is no statistically significant difference in the LEP of the two cases.
- Alternative Hypothesis H_{A3}: There is a statistically significant difference in the LEP of the two cases.

Table 2.3 shows the results of the test. Since the p-values of DEP and NEEP are less than 0.05, we reject the null hypotheses (H_{01} and H_{02}) and accept

Table 2.3: Impact of modelling tool and collaboration on DEP, NEEP and LEP (Mann-Whitney tset)

Data	Mann-Whitney U	sig. 2-tailed
DEP	52.000	0.001
NEEP	42.000	0.000
LEP	125.000	0.249

Table 2.4: Impact of the size of models (Kruskal-Wallis test)

Data	Chi-square	df	p-value
N of Classes	4.151	2	0.126
N of Associations	4.151	2	0.126

the alternative hypotheses (H_{A1} and H_{A2}). In other words, the differences between the mean rankings of DEP and NEEP of the two cases are statistically significant. The p-value of LEP is $0.249 > 0.05$. We cannot reject the null hypothesis (H_{03}), and the difference between the mean rankings of LEP of the two cases is not significant.

We have a statistical evidence to conclude that the DEP and NEEP are affected by the change of both the modeling tool and the number of involved subjects per task. While LEP is not affected by such change.

2.5.3 Impacts of The Topic/Size of The Modeling Scenarios on DEP, NEEP and LEP

We used three different scenarios for the three modeling tasks that were used in EXP1. The scenarios are different in topic, but slightly different in size.

In order to statistically assess the difference in the size, we calculated the number of *classes* and *associations* in each solution created by the students per scenario. After that, we ran Kruskal-Wallis test [114]. The following two hypotheses were formed:

- Null Hypothesis H_0: There is no statistically significant difference between the median number of classes/associations in each solution created for scenario 1, 2 & 3.
- Alternative Hypothesis H_A: There is statistically significant difference between the median number of classes/associations in each solution created for scenario 1, 2 & 3.

The distributions of the number of classes and associations in the solutions of each modeling scenario are reported in Table 2.5. The result of Kruskal-Wallis test is presented in Table 2.4. We cannot reject the null hypothesis (the

Table 2.5: Number of classes and associations in the solutions of each modeling scenario

Scenario	N of classes				N of associations			
	Med.	Q1	Q3	I-Q. R.	Med.	Q1	Q3	I-Q. R.
1	8.00	7.00	8.00	1.00	7.00	6.00	7.00	1.00
2	6.00	4.00	9.00	5.00	5.00	3.00	8.00	5.00
3	6.50	6.00	7.75	1.75	5.50	5.00	6.75	1.75

significance value p = 0.126 > 0.05). In other words, the difference between the number of classes/associations in each solution created for scenario 1, 2 and 3 is not significant.

Results for Q4: We cannot assess the impact of the size of the modeling task (scenario) on DEP, NEP and LEP, as we have evidence that the modeling tasks are not statistically different in size.

At this point we only study the impact of the topic of the modeling scenarios on DEP, NEEP and LEP. In particular, we want to asses if there is any statistically significant difference in the mean of DEP, NEEP and LEP between the three modeling scenarios. To this end, the following set of hypotheses were formed:

- Null Hypothesis H_{01}: There is no statistically significant difference in DEP of the three scenarios.
- Alternative Hypothesis H_{A1}: There is statistically significant difference in DEP of the three scenarios.
- Null Hypothesis H_{02}: There is no statistically significant difference in NEEP of the three scenarios.
- Alternative Hypothesis H_{A2}: There is statistically significant difference in NEEP of the three scenarios.
- Null Hypothesis H_{03}: There is no statistically significant difference in LEP of the three scenarios.
- Alternative Hypothesis H_{A3}: There is statistically significant difference in LEP of the three scenarios.

The normalities of DEP, NEEP and LEP were checked using Shapiro-Wilk test [115]. Table 2.6 shows the p-values of the test. The p-values are less than 0.05, and the data are not normally distributed.

Table 2.6: Normality Test results

Data	Shapiro-Wilk (p-value)
DEP	0.046
NEEP	0.027
LEP	0.000

Table 2.7: Impact of the topic of the scenario on DEP, NEEP and LEP (Kruskal-Wallis test)

Data	Chi-square	df	p-value
DEP	1.040	2	0.595
NEEP	1.630	2	0.443
LEP	4.325	2	0.115

Having non-normally distributed data, we applied the *non-parametric* Kruskal-Wallis test [114]. Table 2.7 shows the results of the test in detail. Since the significance values of DEP, NEEP and LEP are > 0.05, we cannot reject the null hypotheses. In other words, the differences between the mean rankings of DEP, NEEP and LEP of the given three modeling scenarios are not significant.

Furthermore, we compared the differences in the mean value of DEP, NEEP and LEP between every pair of scenarios using Mann-Whitney test [116]. The results reported in Table 2.8 prove that the difference in the mean rankings of DEP, NEEP and LEP is not significant between every pair of the three scenarios (the critical level of significance is $0.05/3 = 0.0167$).

Results for Q5: We have statistical evidence to conclude that the DEP, NEEP and LEP stay the same through different-in-topic modeling tasks.

2.5.4 Subjects Questionnaire

This subsection resumes the feedback gathered from the participants involved in the two experiments. These feedback (presented in Table 2.9) complement the results, and are discussed in Sections 2.6 and 2.7.3.

2.6 Discussion

With the increasing popularity of agile-approaches in software development there has been a reduced commitment from software development projects to

Table 2.8: Impact of topic between every pair of scenarios (Mann-Whitney test)

	DEP (sig. 2-tailed)	**NEEP** (sig. 2-tailed)	**LEP** (sig. 2-tailed)
S1 - S2	0.408	0.316	0.168
S1 - S3	0.343	0.244	0.046
S2 - S3	0.949	0.733	0.394

Table 2.9: Obtained feedback via the questionnaire

Feedback	**Results**				
	Experiment	Med.	Q1	Q3	I-Q. R.
Expertise in software modeling	EXP1	2	1	3	2
	EXP2	3	2	3	1
Experience in using UML	EXP1	2	1	2	1
	EXP2	2	2	3	1
Clarity of the scenarios	EXP1	4	4	4	0
	EXP2	4	3	4	1
Usability of the modeling tool	EXP1	4	3	4	1
	EXP2	3	2	3	1

modeling. Partially this view seems motivated by the seeing of modeling as an activity that produces 'documentation' (rather than 'working code'). Our study shows that a significant part of the effort dedicated to modeling is spent on thinking about the design. Even though the actual impact of this needs to be further assessed, we believe that this thinking about the design is valuable.

For EXP1, our results show that *at least* 56% of the modeling effort is spent on *design*. Whereas for EXP2, *at least* 74% of the modeling effort is *design*. Our assumption is that the participants did not make any *design* effort while expressing the model in UML notation as well as doing layout.

One threat to the interpretation of our experiment is whether or not design-thinking actually happens concurrently with notation expression or layouting. Given the small size of the layout effort, this would only have a small impact on our interpretation. If one believes these cognitive tasks overlap, then the interpretation of our experiment should be that there is indeed more design thinking - i.e. we have found a lower bound on it through our study. Consequently, the percentages that we found are minimum, and may increase as the effort on *designing* overlaps with the effort on the other two activities (notation expression and layouting).

The *notation expression* effort is on average 41% and *layouting* is on average 3%. These efforts may actually represent a cost of UML modeling. This cost may be seen as an investment in the communication-value of documentation within a team. In order to understand whether the cost of notation expression

and layouting is inevitable or not, we may investigate the impact of the usability of modeling tools on the modeling process. Although the participants of EXP1 used a modeling tool (WebUML) different from the one used in EXP2 (Papyrus), we could not reveal the impact of the modeling tool on the modeling effort. This is because the modeling solution of scenario 2 of EXP1 was created by participants individually, while the same scenario was modeled in by 4 participants collaborating in a team in EXP2. So, the difference in the DE, NEE and LE between the two settings could be due to the type of the modeling tool or the number of assigned participants per modeling task. Observing the perceived usability of the modeling tools by the participants of the two experiments (see Table 2.9), it might be that the difference in DE and NEE between EXP1 and EXP2 is due to the factor of collaboration, i.e. design discussions. As a future work, we aim to isolate the impact of these two factors (modeling tool and collaboration) on the modeling effort, as well as investigate better modeling-tool support [117].

2.7 Threats to Validity

2.7.1 Construct Validity

We benefited from the fact that we performed the experiment in a controlled environment (instead of as a homework assignment): 8 students created the model and its copy in parallel. We eliminated these cases from our data set because we could not explicitly calculate the modeling and copying efforts. For replication of this research, a lesson learned is to instruct students not to do *model* and *copy* simultaneously.

We did not aim at maximizing realism and focused on class diagrams for various reason. We wanted to: (i) ask simple tasks based on a well-known notation; (ii) reduce confounding factors and thus keep more control over the experiment compared to drawing multiple types of diagrams; and (iii) have a preliminary result that can validate our vision.

2.7.2 Internal Validity

It could be possible that thoughts wander off into unrelated territories. The following reasons limit the impact of this phenomenon: Firstly, the experiment was performed in a controlled environment. Supervisors walked around the room. They monitorred that there were no distractions like coffee drinking or going to the bathroom. Also, they observed that the participants were indeed working on the task behind their computers. Suppose that indeed some wandering of thoughts happens, then our expectation is that this happens more or less equally in the first phase (*modeling*) and the second phase (*copying*). In

this case at least the relative ratio between these tasks should not be affected much.

In the redrawing of a copy of the previously created solution, participants in the study may have benefited from a *learning effect*. This could have led to the *notation expression effort* being a less than the *notation expression effort* in the initial creation of the UML models. We think this affect is small, and that a more detailed analysis of notation creation activities can lead to a quantification of this learning effect. However in *EXP2*, we tried to mitigate this factor by asking each group to copy the created solution of another group.

Moreover when performing the second modeling task (second scenario) in EXP1, participants may have benefited from a *learning effect* as well as suffered from the *fatigue* of performing two modeling tasks consecutively. We do not think there is a large *learning effect* in the use of the tool, because participants have been trained in using the tool both before starting and as part of this assignment. Hence they already have a reasonable fluency in the tool at the start of the first scenario. We do think the affect of *fatigue* is small because the modeling scenarios are simple, and the required time to complete one scenario is not too much (around 25 minutes).

2.7.3 External Validity

Complexity of the scenarios. The scenarios were kept simple and clear so that the students can easily understand and complete the tasks in the time of the experiment. In industry settings, modeling tasks can be much more complicated in term of size, terminology, languages, level of details, etc. which we could not cover in our study. This is a limitation on the generalizability of the findings of this paper when it comes to real-world cases. However, we consider this threat as acceptable for this preliminary investigation. We are working on studying larger scenarios to increase the generalizability.

Participants and their modeling expertise. The participants involved in our experiment may not represent the general population of modeling practitioners. Moreover, the modeling expertise of our participants is relatively homogeneous. This limits us from generalizing our findings to other subjects (i.e. experts, professional software architects, industrial practitioners in the field). Indeed, familiarity with the modeling tool and experience of designing may result in different DE, NEE and LE percentages. We consider our findings as a basis to extend our study to larger community of modeling practitioners.

2.8 Conclusion and Future Work

In order to better understand the effort involved in using software models in software development, we introduced in this paper the distinction between

design, *notation expression* and *layout* efforts. Subsequently, we defined and ran two experiments in which we measure how much effort each of these activities takes – both in absolute effort and as a percentage of the total effort spent on creating class diagrams in a simple student assignment. From these experiments we conclude that UML modeling should not be considered so very costly, because it triggers design thinking. According to our results, the effort spent on thinking about synthesizing the design takes *at least* 56% of the total modeling effort.

One implication of this research is that projects that create models concur at least with significant thinking about the design. This aligns with an earlier finding that developers report that creating design models in the early stage of a software development projects, leads to better modularity of the design [104].

Future work: This line of research can be extended in many directions. In order to increase the external validity, we would like to obtain data from larger and/or industrial projects about their modeling effort. We have started looking into two projects where teams of 8 students work for 3 months full time on a software project. Another extension that is of interest is to study the effect of usability of modeling tools on the time spent on subtasks of modeling. This would highlight if tool complexity is a major factor in adoption of modeling. Complementary questions would be to: (i) explore how much effort is involved in maintaining models up-to-date in documentation throughout a project, and (ii) study the impact of the *designing* and *modeling* on the speed and quality of software development.

Chapter 3

OctoUML

(B) R. Jolak, B. Vesin, M.R.V. Chaudron "OctoUML: An Environment for Exploratory and Collaborative Software Design"
In the 39th International Conference on Software Engineering Companion (ICSE-C), pp. 7-10. 2017.

Abstract

Software architects seek efficient support for planning and designing models at multiple levels of abstraction and from different perspectives. For this it is desirable that software design tools support both informal and formal representation of design, and also support their combination and the transition between them. Furthermore, software design tools should be able to provide features for collaborative work on the design. OctoUML supports the creation of software models at various levels of formality, collaborative software design, and multi-modal interaction methods. By combining these features, OctoUML is a prototype of a new generation software design environment that aims to better supports software architects in their actual software design and modelling processes.
Demo video: `https://youtu.be/fsN3rfEAYHw`
OctoUML Project: `https://github.com/Imarcus/OctoUML`

Keywords: software design; modelling notations; multi-modal interaction; collaborative design; user experience; UML

3.1 Introduction

Designing software consists of exploring design problems, discussing solutions and creating software models as design artifacts. Such artifacts provide a bridge between problem and software implementation by describing user's needs as well as the product to be developed. As software systems are gaining increased complexity, the importance of efficient software design tools is also increasing. Software models change frequently and are quite often updated by many designers simultaneously [118]. These models should present a description of complex systems at multiple levels of abstraction and from a different perspectives. Therefore, it is crucial to provide software design tools that give possibilities for efficient and collaborative development as well as options for multi-modal interaction.

Modelling tools can be classified into two groups: informal and formal [84]. We mean by informal any tool that supports informal design in the sense that it does not constrain the notation used. Indeed, informal tools are preferred for their flexibility as well as the role that they play in unleashing designers' expressiveness. Examples of such tools are whiteboards, paper and pencil. While we mean by formal any tool that support one or few formalized notations. Typical examples are UML CASE-tools (e.g. Rational Rose, Enterprise Architect, Papyrus, StarUML, etc.). Formal tools are usually used for code-generation and/or documenting purposes.

During early design phases, software designers often use informal tools (e.g. whiteboards) to sketch their thoughts and compare design ideas. Once the designers settle on one possible solution, they proceed to create a formal version of the sketchy design. In particular, they move from the whiteboard, start-up the computers, run a formal tool (a CASE-tool), and re-enter the solution that has been created previously during the early design phase. So there is a gap between informal designing in early software design phases and formal design and documentation practices in subsequent development. To bridge this gap, we present *OctoUML*, a software design environment that supports exploratory and collaborative design meetings. OctoUML provides means to allow the creation of both sketchy hand-drawn elements and formal notations simultaneously. Moreover, it allows the transformation of sketchy designs into formal notations.

Oviatt and Cohen [119] illustrated the importance of multi-modal systems in reshaping daily computing tasks and predicted their future role in shifting the balance of human-computer interaction much closer to the human. We enabled OctoUML to support multiple modes of interaction including mouse, keyboard, touch/multi-touch using fingers and styluses, sketching, and voice modality.

More often than not, the process of software design involves several designers working on the same project simultaneously. This could also occur in user-

centered design situations where users are involved in the design process. We implemented OctoUML to support design collaborative sessions, both *in-situ* (via the adoption of multi-touch technique) and at a distance *"remotely"* (by using a client-server paradigm). OctoUML can be run using a number of input devices ranging from desktop computers over large touch screens to large interactive whiteboards.

The paper is organised as follows: the related work is presented in section two. Further information on OctoUML, its architecture and features, and the performed evaluation are reported in section three. The future objectives and concluding remarks are presented in the last section (section four).

3.2 Related Work

Several studies proposed different approaches to enhance the software design process. Mangano et al. [71] identified some behaviors that occur during informal design. In particular, designers sketch different kind of diagrams (e.g. box and arrow diagrams, UI mock-ups, generic plots, flowcharts, etc.) and use impromptu notations. The authors implemented an interactive whiteboard system (called *Calico*) to support these behaviors and identified some ways where interactive whiteboards can enable designers to work more effectively.

Wüest et al. [72] stated that software engineers often use paper and pencil to sketch ideas when gathering requirements from stakeholders, but such sketches on paper often need to be modelled again for further processing. A tool, *FlexiSketch*, was prototyped by them to combine free-form sketching with the ability to annotate the sketches interactively for an incremental transformation into semi-formal models. The users of *FlexiSketch* were able to draw UML-like diagrams and introduced their own notation. They were also able to assign types to drawn symbols. Users liked the informality provided by the tool, and had the will to adopt it in practice.

Magin and Kopf [120] created a multi-touch based system allowing users to collaboratively design UML class diagrams on touch-screens. They have also implemented a new algorithm to recognize the gestures drawn by the users and to improve the layout of the diagrams. However, their tool does not allow for informal freehand sketching of arbitrary notations.

Lahtinen and Peltonen [121] presented an approach to build speech interfaces to UML tools. The authors set up a spoken language to manipulate the UML models, and built a speech control system (*VoCoTo*) integrated with a CASE-tool (*Rational Rose*). They stated that speech recognition is applicable to be used to enhance the interaction with UML tools.

Table 3.1 summarizes the main supported functionalities by OctoUML and illustrates the differences to the related work.

Table 3.1: Comparison between OctoUML and the related work.

Related Work	Informal & formal notations	Interaction Modalities	(Multi-Touch, Remote Control)
Calico	informal hand-drawn notations	mouse, keyboard and touch	(no, no)
Flexisketch	informal hand-drawn notations	mouse, keyboard and touch	(no, no)
Magin&Kopf	formal notations creation via gestures	touch-based	(yes, no)
VoCoTo	formal notations	mouse, keyboard and voice	(no, no)
OctoUML	creation and mix of informal and formal notations simultaneously	mouse, keyboard, single touch, multi-touch, and voice	(yes, yes)

3.3 OctoUML

In a previous work [84], we presented our vision for a new generation software design environment. To realize our vision, we developed a prototype called OctoUML [117]. OctoUML is a software design environment that supports exploratory and collaborative software design. It is used to create and organize diagrams as well as supports their modification and evolution. Firstly, we illustrate the architecture of OctoUML. Secondly, we describe the main functionalities that are supported by OctoUML (sections B and C). Later on, we provide a scenario showing how such functionalities could support the design process. Lastly, we provide some details on OctoUML evaluation.

3.3.1 OctoUML's Architecture

The key architectural components of OctoUML are presented in Figure 3.1. The environment contains three major components: *UI component*, *Data component* and *Services*. The current version of the system offers only the *UI* and *Data* components. Additional services will be added during future development. The *UI component* consists of: *Presentation manager* and *Input unit*. The *Presentation manager* provides means for performing stylus or touch-based input commands on devices being used. *Drawing layers* include support for both informal and formal modelling layers. The *Command tools* are responsible for transferring the inputs from users to different controllers. The *Graph controller* allows switching between different input techniques with combining of multiple layers. The *Input unit* is responsible for processing different inputs. In particular, a *Sketch recognizer* is provided to recognize and transform informal models into formal concepts, and hence allows to maintain and transfer the designs for further processing tasks. A *Multi-touch*

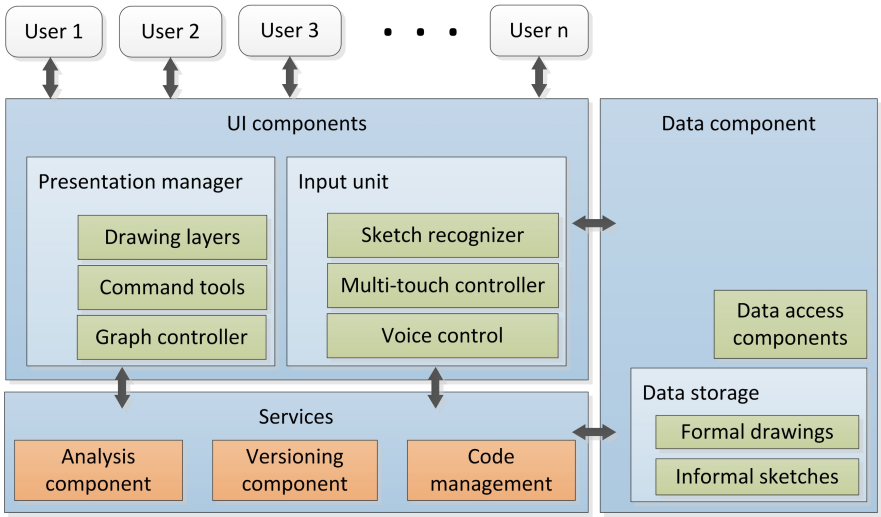


Figure 3.1: Architectural Components of OctoUML

controller captures and coordinates the inputs from different touch-points. All the program data are saved and stored in the *Data component*. Our tool uses a set of data structures to manage and maintain the sketched elements and formalized designs.

3.3.2 Informal and Formal Notation

Whiteboards (or any informal tools e.g. paper and pen) are used during early software design phases because of their flexibility and immediacy, but also becuase they do not constrain the notation being used. Informal notations (e.g. *sketches*) can be used to express abstract ideas representationally, to allow checking the entirety and the internal consistency of an idea as well as to facilitate development of new ideas [31]. Furthermore, informal notations can have a very close mapping to the problem domain. However, the informal notations often need to be formalized in order to allow their manipulation and process e.g. sharing, code generation or documentation.

Modelling tools should not constrain designers to create only some specific notations. Furthermore, they should maintain the characteristics of formal tools in their support of design transfer and persistence [84].

OctoUML allows the creation of both hand-drawn informal sketches and computer-drawn formal elements (currently UML class and sequence models) on the same canvas simultaneously (Figure 3.2). OctoUML bridges the gap between early software design process, when *informal* tools are typically used,

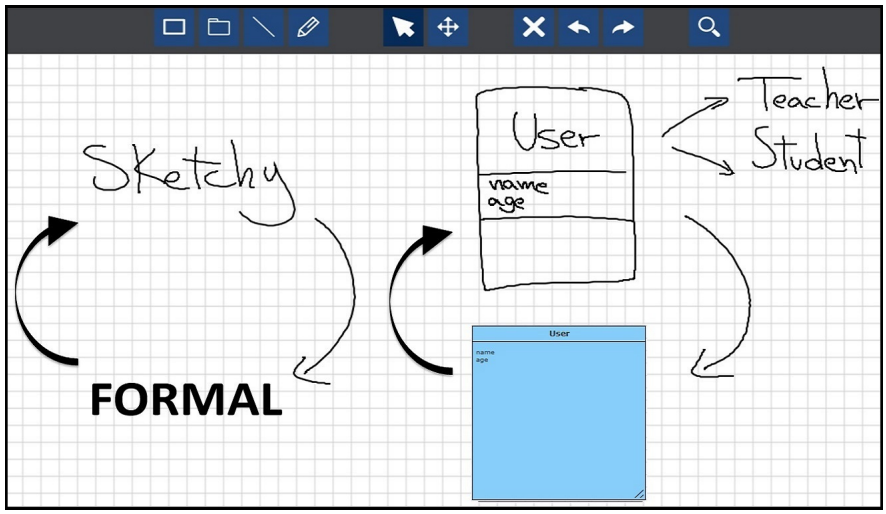


Figure 3.2: Combination of different notations on the same canvas

and later documentation and formalization process, when *formal* tools are used. Beside supporting the creation of software models at different levels of formality, OctoUML is equipped with a *Sketch recognition unit* which enables sketch formalization. In particular, OctoUML allows the transformation of models from informal to formal and vice versa at any time during the modelling session. Furthermore, we adopted a layering technique by which the informal notations belong to one layer that we call the *informal layer*, and the formal notations belong to another layer that we call the *formal layer*. The user can then select to see the layers in combination or isolation.

3.3.3 Interaction Modes and Collaboration

The usability of current CASE tools is a common source of criticism [13]. The interaction with such tools is often based on using the mouse and keyboard. Other modes of interaction (e.g. touch, gesture and voice) could be more natural and intuitive. In order to improve the user experience of OctoUML and increase its accessibility, the interaction modalities of OctoUML are enriched by providing a *voice-commands* recognition component capable of transforming designers' voice-commands into control actions.

The process of software design often involves more than one designer working on the same project simultaneously. OctoUML promotes collaborative design by adopting a *multi-touch* technique and supporting *remote collaboration*. Next, we provide more details on the supported functionalities:

- *Multi-touch* is an interaction technique that permits the manipulation

of graphical entities by more users at the same time. Our tool allows multiple users to design diagrams simultaneously by performing simple touch gestures.

- In order to improve the user experience, we integrated a *voice-commands* control component within the *Input unit*. The component is capable of handling the most commonly used functions during the design process. Thus, users can use voice commands in order to create and edit elements of software diagrams.

- To open up new opportunities for interactive collaborative design, our tool supports *remote collaborative* sessions between geographically distributed teams. One team of designers can run a server instance of OctoUML, whereas another team can join the session as client connecting to the server. Video calls and chatting tools will be integrated in order to support the joint design sessions.

3.3.4 Design process in UctoUML: A Scenario

Figure 3.3 illustrate the design process in OctoUML. Activities that are currently supported by OctoUML are distinct in green. Let us think about the following scenario: a group of software designers meet to explore and discuss design ideas of a specific software product. The designers start with the creation of some informal sketchy designs using OctoUML being deployed on a large interactive whiteboard. After that, the designers proceed with a selective transformation of some informal sketches into a formal model. Later on, the created model is analyzed to check possible flaws and performance bottlenecks. Finally, the model is saved and uploaded to a version control repository. The designers meet again (on-site or from different locations) when new requirements come out or having earlier requirements exposed to changes. They fetch the design that was previously shared on the version repository, update the design, and commit a new version that is now compliant to the new requirements.

3.3.5 Evaluation

Two *user studies* were performed to evaluate OctoUML. In both studies, the participants had to do a modelling task using OctoUML, answer a System Usability Scale (SUS) questionnaire [122], and participate into semi-structured interviews. The first study involved fourteen software engineering students (ten PhD and four M.Sc. students) and two post-doc researchers. The main purpose of the first study was to evaluate the usability of OctoUML as well as to investigate whether supporting the mix of informal and formal notation could support the design process. OctoUML got an average SUS-score of 78.75 which is a high usability score according to [86]. The participants stated that

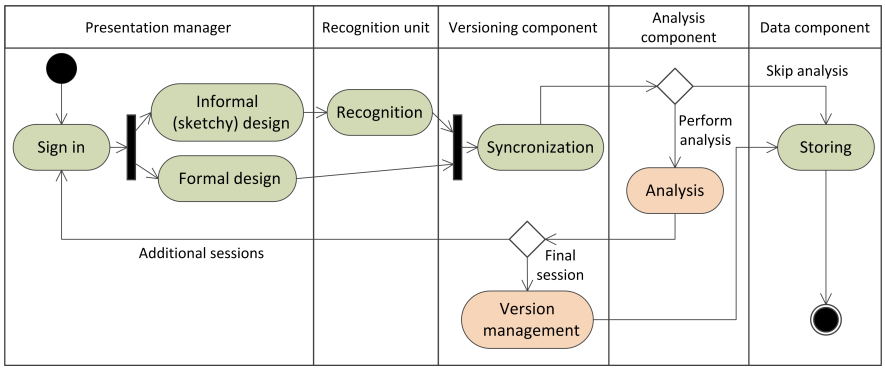


Figure 3.3: Design process in OctoUML

informal notations could be valuable artifacts beyond being just explorative means. They also stated that such notations support designers' activities in understanding the problems and communicating ideas. Figure 3.4 shows the feedback from the participants regarding the use of informal and formal notations within OctoUML.

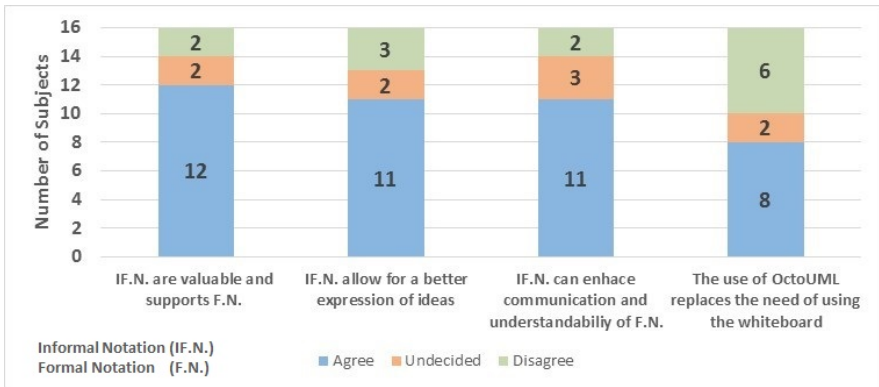


Figure 3.4: User study I: informal vs. formal notations

The second study involved fourteen participants (three PhD and eleven M.Sc. software engineering students). The main purpose was to evaluate the learnability and usability of OctoUML as well as the role of the voice interaction modality in enhancing the user experience and supporting the software design process. OctoUML got a SUS-score of 74.6 which can be considered a quite good usability score [86]. The majority of the participants stated that it was easy to learn and use the different functionalities of OctoUML (including the voice interaction modality), see Figure 3.5. Furthermore, the voice interaction

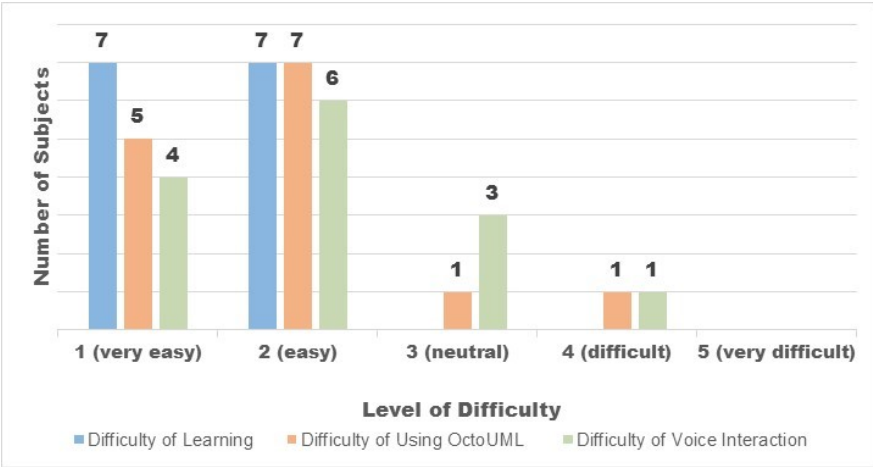


Figure 3.5: User study II: usability and learnability of OctoUML

modality was perceived helpful in overcoming non-ergonomic tasks e.g. typing via a keyboard.

3.4 Conclusion and Future Development

In this paper we presented OctoUML, a prototype of a new generation software design environment for collaborative software design. It provides support for mixing informal hand-drawn elements with formal notations. Moreover, it supports different input methods and interaction modalities.

OctoUML combines the advantages of both informal tools e.g. interactive whiteboards and formal tools e.g. CASE tools, and therefore is able to bridge the gap between early software design process (when designers often sketch their ideas) and formalisation/documentation process. OctoUML was evaluated by conducting two user studies and involving thirty participants in total. The main goal was to get feedback on the viability and usability of OctoUML. The results show that the participants enjoyed their experience with OctoUML and had a positive perception regarding its usability.

The current architecture of OctoUML allows future expansions of the system with additional functionalities. The goal is to implement and incorporate additional features in the subsequent versions of the system:

– *Analysis component.* It will perform software model analysis. This tool will be used to automatically evaluate the created software models to detect general design flaws, security flaws and performance bottlenecks.

- *Versioning component.* The purpose is to provide a repository for keeping track of the version history of stored models, and the ability to observe changes that are made to specific artifacts in the environment. The system should also be able to resolve conflicts when two users change the same model data. Such component would increase the potential for parallel and distributed work, improve the ability to track and merge changes over time, and automate management of revision history. It would also allow multiple designers to work concurrently, supporting tight collaboration and a fast feedback loop.
- *Code management.* Models and code must be combined throughout the development process. Users will be able to generate code from formalized UML class diagrams as well as view models and codes side by side and jump between editing one and keeping the other synchronized.

Chapter 4

Model-Based Software Engineering: A Multiple-Case Study on Challenges and Development Efforts

(C) R. Jolak, T. Ho-Quang, M.R.V. Chaudron, R.R.H. Schiffelers "Model-Based Software Engineering: A Multiple-Case Study on Challenges and Development Efforts"
In the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 213-223. 2018.

Abstract

A recurring theme in discussions about the adoption of Model-Based Engineering (MBE) is its effectiveness. This is because there is a lack of empirical assessment of the processes and (tool-)use of MBE in practice. We conducted a multiple-case study by observing 2 two-month MBE projects from which software for a Mars rover were developed. We focused on assessing the distribution of the total software development effort over different development activities. Moreover, we observed and collected challenges reported by the developers during the execution of projects. We found that the majority of the effort is spent on the *collaboration and communication* activities. Furthermore, our inquiry into challenges showed that tool-related challenges are the most encountered.

Keywords: Software Engineering; Model-Based Engineering; Effort Distribution; Modeling Tools; MBE Challenges; Case Study Design.

4.1 Introduction

Models provide effective means for supporting the communication between stakeholders, and serve as specification for the implementation of software systems. Model-Based Engineering (MBE) is a software development approach in which models play an important- *central* role [123]. MBE aims to increase the abstraction level and aims to promote the automation of the development process [3].

Empirical assessment of the use and process of MBE is scarce. The adoption of MBE is still debated in practice. On the one hand, MBE has been applied effectively in several application sectors, e.g., embedded systems [4] and telecommunication [23]. Furthermore, by focusing on practitioners' experiences and perceptions, several studies claimed that the adoption of MBE helps to (i) improve the productivity of the developing teams by increasing the abstraction level, (ii) enhance the quality of the software and (iii) support its maintainability [4, 21, 87]. On the other hand, some practitioners consider MBE as a time-consuming and unproven approach that merely complicates matters [3].

4.1.1 Rationale

Generally, the field of software engineering perceives a discrepancy between empirical software engineering findings and developers' (*a priori*) beliefs and opinions, which are often based only on personal perspectives on the development processes. Devanbu et al. [6] suggested that more in-depth studies that address the interplay of belief and evidence in software practices are needed. The same issue is pointed out by Ralph [124] who differentiated between two paradigms of software development research: *Empirical* and *Rational*. Empiricists believe that knowledge can only be justified by sense experience and observation. In contrast, rationalists accept that some knowledge can be justified by observation, but claim that other knowledge is justified by reason or intuition. Ralph claims that the rational paradigm continues to dominate the software engineering standards and approaches: many developers and researchers hold beliefs that are incongruous with empirical evidence. This, according to Ralph, would undermine the software engineering community's scientific credibility.

4.1.2 Objective and Contribution

This study contributes to the body of knowledge on the process and use of MBE in practice. In particular, we conducted a multiple-case study [78] by analyzing and discussing empirical data collected from 2 two-month MBE projects carried out at the Technical University of Eindhoven, the Netherlands. The main contributions of this paper are two-fold:

- Firstly, we shed light on the distribution of development efforts in MBE. The resulting observations on effort distribution could lead to improved MBE project planning and organization (e.g., resource allocation and risk management), which in turn could lead to cost reduction.
- Secondly, we report and further analyze different challenges to the process and use of MBE in practice. Exposing such challenges would make them a candidate subject for research that are concerned with MBE process improvement. Moreover, understanding and providing ways to overcome these challenges could bring a significant impact to the effectiveness and efficiency of the MBE approach.

In this paper, we address the following research questions:

- **R.Q.1** How is the total effort spent on MBE distributed over different development activities?
- **R.Q.2** How is the effort spent on different MBE development activities distributed over time?
- **R.Q.3** How large is the portion of collaborative work in MBE projects?
- **R.Q.4** What are the challenges that affect MBE in practice?
- **R.Q.5** How are the challenges that affect MBE distributed over project time?

The remainder of this paper is organized as follows: in Section 4.2, we consider and discuss the related work. We describe the case study design in Section 4.3. We present and discuss the results in Section 4.4. We discuss the threats to the validity of this study in Section 4.5. Finally, we conclude and discuss the future work in Section 4.6.

4.2 Related Work

In this section, we review the published work on: (i) measuring effort distribution between MBE development phases, and (ii) challenges encountered when adopting MBE.

4.2.1 Effort distribution in MBE

Distribution of effort in software engineering processes is largely researched in the context of estimation and planning of software projects [125]. Several practitioners studied the effort required for different software development activities, and provided rules of thumb such as the "40-20-40" rule of Pressman [126], that is 40% on analysis and design, 20% on coding and 40% on integration and

testing. Other rules of thumb were provided by: Ambler [127], Boehm [128], Boehm et al. [129], Brooks [130] and Zelkowitz [131].

In his text book, Sommerville [132] estimates effort distribution by measuring cost units in different development activities, i.e., 15 units on specification, 25 units on design, 20 units on development/implementation and 40 units on testing.

Yang et al. [133] empirically studied development effort distribution of 75 projects from 46 software organizations from the China Software Benchmarking Standard Group (CSBSG) database. The development approaches defined in CSBSG database roughly follow the waterfall model, including planing, requirements, design, coding, testing and transition. The following mean efforts over each development phases are reported: 16.14% for planning and requirements, 14.88% for design, 40.36% for coding (including unit test and integration), 21.57% for testing (system testing), 7.06% for transition (including installation, acceptance test and user training).

Recent works including the one by Papatheocharous et al. [134] studied effort distribution based on projects obtained from the International Software Benchmarking Standards Group (ISBSG) R10 dataset [135]. The six development phases declared by ISBSG are: planning, specification, design, build, test and implementation. The mean efforts spent on each development phase are reported as follows: 8.2% for planning, 7.9% for specification, 11.9% for design, 36.8% for developing and building, 15.5% for testing, 5.6% for implementation and 14.0% for unphased activities.

On the basis of data collected from 20 industrial software development projects, Heijstek and Chaudron [136] reported effort distribution over various disciplines, defined by the Rational Unified Process (RUP), in MBE. The following effort distribution was reported: 11% for analysis & design, 8% for requirements analysis, 12% for testing, 38% for implementation, 13% for project management, 4% for change & configuration management, 3% environment, 2% for deployment, 9% for others choices. This is, according to the authors, surprisingly similar to the RUP Hump chart, and thus underlines the similarity between MBE and traditional development approaches. To the best of our knowledge, this study is so far the only one that investigates effort distribution in MBE projects.

4.2.2 Challenges in MBE

Although MBE claims many potential benefits, e.g., gains in productivity, portability, maintainability and interoperability [19–21], its adoption has been facing a number of challenges. These challenges are discussed in academic forums and empirically investigated in a number of industrial cases.

Van Der Straeten et al. [137] summarize outcomes of a plenary session at the MODELS'08 workshop on "Challenges in Model-Driven Software Engineering"

where participants discussed challenges in the field of MDE. Discussed challenges included: management of models quality, lack of focus on modeling process and models at run-time, and insufficient MDE tool-support.

Lately, Mussbacher et al. [22] reflected opinions of 15 MDE experts on the biggest problems with MDE technologies over the last 20 years. The authors highlighted that tools *usability* and *adoption*, people's diverse perception of MDE, inconsistencies between software artifacts, and lack of fundamentals in MDE are considered as hindrances to MBE adoption.

Baker et al. [23] discussed experiences with MBE/MDE at Motorola over a time span of almost 20 years. A number of challenges were reported, such as poor tools and generated code performance, lack of integrated tools, and lack of scalability.

Hutchison et al. [5] analyzed 250 survey-responses and 22 interviews, as well as did on-site observations of MDE. They found that the main challenges to MDE adoption are significantly related to Domain-Specific Languages (DSLs) and MDE tools, as well as to organizational factors and human training issues. Based on a survey involving 113 software practitioners, Forward and Lethbridge [138] reported common problems with model-centric development approaches. These problems are related to inconsistency of models over time, model interchange between tools, and heavyweight modeling tools.

Similarly, by surveying 155 Italian software professional, Torchiano et al. [18] considered lack of competencies and supporting tools as the main show stoppers preventing altogether the adoption of modeling and model-driven techniques.

In the embedded systems domain, Liebel et al. [17] analyzed survey-responses from 122 professionals working with MBE, and considered that interoperability between (MBE/MDE) tools as a main challenge to MBE adoption. Moreover, other factors such as, high effort to train developers and tools (poor) usability, were also identified as secondary MBE challenges.

While the above-mentioned studies help in exploring challenges, there are a number of other studies which focus on specific challenges, especially tool-related ones. By performing a series of interviews with 20 engineers and managers at General Motors, Kuhn et al. [139] identified five points of friction in MDE. All of them are related to MDE tools. Similarly, by analyzing a total of 39 interviews with industrial practitioners, Whittle et al. [14] identified a taxonomy of technical, social and organizational issues related to MDE tool use in practice. Addressing such issues together with modeling tools-related issues identified by this study, would probably ameliorate the effectiveness and efficiency of MBE.

4.3 Case Study Design

Yin defined case studies as empirical inquiries to perform a deep investigation of a particular phenomenon, where the boundary between the phenomenon and

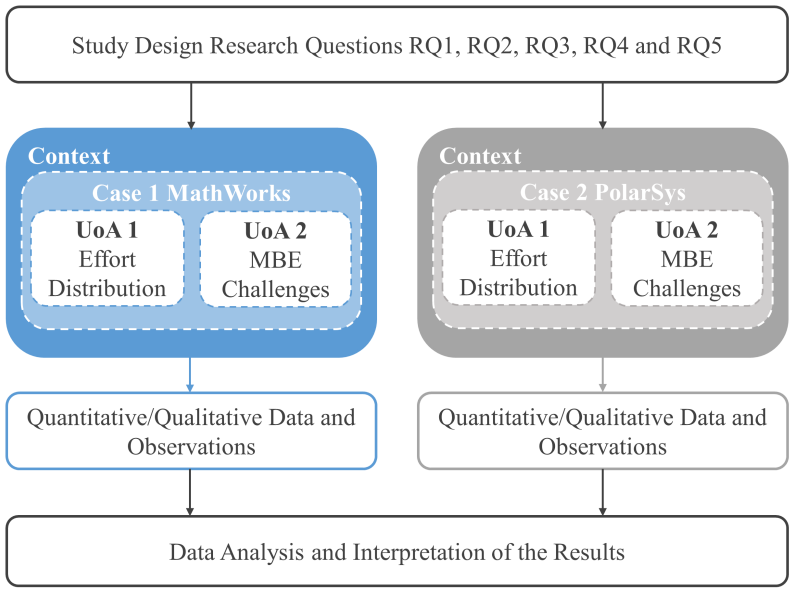


Figure 4.1: Case Study Design.

its real-life context cannot be clearly specified [78]. Our multiple-case study is an *exploratory- inductive* empirical research [79] conducted to identify patterns in observations, seek new insights, and generate ideas and hypotheses for new research. Figure 4.1 shows the design of our multiple-case study. Further details regarding the case study design will be presented in the following subsections.

4.3.1 Purpose and Cases

Multiple-case studies are regarded as being more robust than single-case studies, and the evidence from multiple cases is often considered more compelling [140]. The intention of the study is to explore the effort distribution over different MBE development activities. Moreover, the study seeks to identify challenges and impediments that could hinder the use of MBE in practice. In particular, two cases were examined:

(A) *MathWorks*: MBE of the software of a Mars rover using MBE tools as provided by MathWorks technologies, e.g., Matlab and Simulink.

(B) *PolarSys*: MBE of the software of the same rover using MBE tools as provided by PolarSys open source technologies, e.g., Papyrus and Capella.

4.3.2 Units of Analysis

The multiple-case study is embedded [78], with two Units of Analysis (UoA):

(A) *Effort Distribution*: Analysis of the distribution of the total MBE effort over different development activities over time.

(B) *MBE Challenges*: Analysis of the challenges that could hinder the adoption of MBE in practice.

4.3.3 Propositions

The two cases are selected to predict possible contrasting results (*theoretical replication*) on MBE challenges and development efforts by altering one condition: the used MBE tools (MathWorks vs. PolarSys). Furthermore, the findings of the this study will be compared to those of other related work in order to find out any eventual supportive similarities or contradicting differences.

In particular, based on the related work we state the following two propositions:

- *Proposition A*: We propose that the distribution of development effort over different MBE activities follows the rules-of-thumb e.g., the "40-20-40" rule. *If* our findings are not compliant, *then* a deeper investigation of the MBE approach is needed in order to understand why the effort distribution deviates from the standard rules.

- *Proposition B*: We propose that poor tool-support is the most frequently reported challenge that affect the adoption of MBE in practice. *If* the perceived challenges in our study do not match, *then* a deeper investigation of the severity of the perceived challenges (both of our study and related work) is needed.

Moreover, for the scope of this multiple-case study, and based on the planned cross-case analysis, we state the following two additional propositions:

- *Proposition C*: We propose that the distribution of the development effort in case 1 matches that of case 2. *If* the distributions do not resemble, *then* the choice of tools and technologies in MBE could affect the development effort. Hence, a deeper investigation of the impact of MBE tools on the development effort is needed.

- *Proposition D*: We propose that similar challenges would be perceived in the two cases. *If* different challenges with different severities are perceived, and if the differences are mainly related to the used MBE tools, *then* we suggest that there is a difference in the maturity between the two technologies (i.e., MathWorks vs. PolarSys).

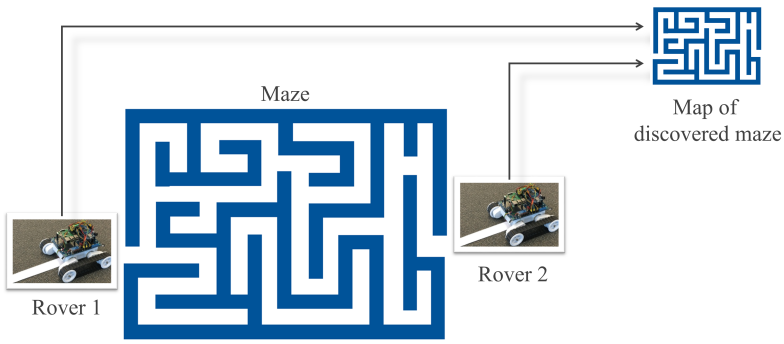


Figure 4.2: Maze-discovering rovers assignment.

4.3.4 Context

The Professional Doctorate in Engineering (PDEng) program of the TU/e aims to train graduates that have (a non-CS) MSc diploma to become professional software engineers. A group of 17 trainees of this program were involved in two projects to develop software for a couple of collaborating concurrently-maze-discovering rovers. The rover system was chosen because of its similarity to the type of software that is developed at the ASML company[1] in Eindhoven.

For both projects, it was mandatory to use MBE as the development approach (e.g., *almost* all software were developed and generated based on, and from, models). The objective of the projects was to develop control software for two rovers that would concurrently discover a single maze of roads as fast as possible. See Figure 4.2. Here, a road consists of a small line of tape with a reflection that differs sufficiently from the underground it is mounted on. The rovers were equipped with IR sensors and a (low-resolution) PID-controller that enabled the rovers to autonomously follow the roads. The rovers had to communicate with each other in order to complete their task in discovering different parts of the same maze. The discovered map of the maze had to be visualized to the end-user during the discovery, and had to be persisted as an end-result.

The trainees were bootstrapped with the hardware and a software API to control the hardware (e.g., motion of the rover), as well as to get the sensors' readings (for e.g., line tracking). The software API was provided at the start of the projects, the hardware itself was provided late in the projects as usually is the case in real-life situations. The main deliverable of the two projects was to produce the rovers' software application. However, in order to test the maze-discovering software application, the trainees needed to develop a software simulator of the hardware. Once the rovers' software application was

[1] `https://www.asml.com`

verified and validated, the simulator was replaced by the actual hardware.

The supervisors of the projects could accept deviations from the initial requirements w.r.t. the development process. That would be possible if and only if such deviations were very well motivated and supported by the trainees.

The group of trainees was organized as follows:

- *Team MathWorks*: Consisted of seven trainees: One team leader responsible of general team tasks as well as team process, risks and design. One design manager appointed to collaborate with the PM on the architecture. One test manager responsible for managing UI tests, unit tests and acceptance tests. One quality manager responsible for code-review for quality assurance and managing documentation (coding and documentation standards). Three developers responsible for modeling, code generation, and manual coding as well as testing. The *MathWorks* team had to develop both software systems (the rover software application and the simulator) using MBE/MDE tools as provided by the MathWorks[2] technologies, e.g., Matlab[3] and Simulink[4].

- *Team PolarSys*: Consisted of six trainees. One team leader, one design manager, one test manager, one quality manager and two developers. These roles had the same responsibilities as described in team *MathWorks*. The *PolarSys* team had to develop the same software applications using MBE/MDE tools as provided by the PolarSys[5] open source technologies, specifically, *Papyrus*[6] and *Capella*[7]. Other tools used by the two teams are reported later in Section 4.4.4.

- *Configuration & Integration Support*: Three trainees given the task to setup, configure and support a continuous development and integration environment for both *MathWorks* and *PolarSys* teams.

- *Project Management (PM)*: Both teams were managed by one trainee, who was responsible of the plan, process, risks, architecture and integration management of the two teams.

The two teams organized themselves in an agile way and had to complete their projects in two months working full-time (i.e., each trainee worked approximately 8 hours per day). The trainees of each team had to meet with the PM and discuss the progress on a weekly basis. Figure 4.3 summarizes the organization of the development teams augmented with size and role information.

[2] `https://se.mathworks.com`
[3] `https://se.mathworks.com/products/matlab.html`
[4] `https://se.mathworks.com/products/simulink.html`
[5] `https://www.polarsys.org`
[6] `https://www.eclipse.org/papyrus`
[7] `https://www.polarsys.org/capella`

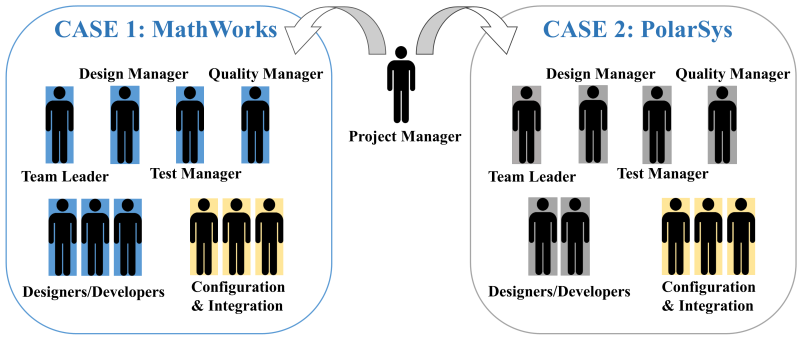


Figure 4.3: Organization of the development teams.

4.3.5 Data collection and Analysis

The data collection consisted of weekly questionnaires as well as developers' time and actions tracking tools. Each week, the developers had to answer a questionnaire[8] in which we collected, amongst others, evidence on: (i) the perceived effort spent on different MBE activities, and (ii) challenges and impediments that affected the development process. The *ProcrastiTracker*[9] software was used to automatically track for each developer which applications and documents were used on their computer and for how long. We also collected the recorded log files produced by this software for each developer on a weekly basis.

We intend to analyze the results by means of *pattern matching* and *cross-case synthesis* [78]. Pattern matching helps to compare an empirically observed pattern with another pattern. When they agree then the pattern is true. Whereas, cross-case synthesis can be used in multiple-case studies to investigate and compare the different cases. Moreover, we intend to use NVivo[10] for qualitatively analyzing the data related to the experienced challenges to MBE approaches.

4.4 Results

In this section, we present the findings of this study together with their interpretation, as well as in relation to the published work and stated propositions. We recall that the findings are based on the considered multiple-case study and its context. Also, as mentioned previously in Section 4.3.5, we recall that we used *pattern matching* and *cross-case synthesis* for data analysis and interpretation.

[8] `http://www.rodijolak.com/pdf/WeeklyQuestionnaire.pdf`
[9] `http://strlen.com/procrastitracker`
[10] `https://www.qsrinternational.com/nvivo`

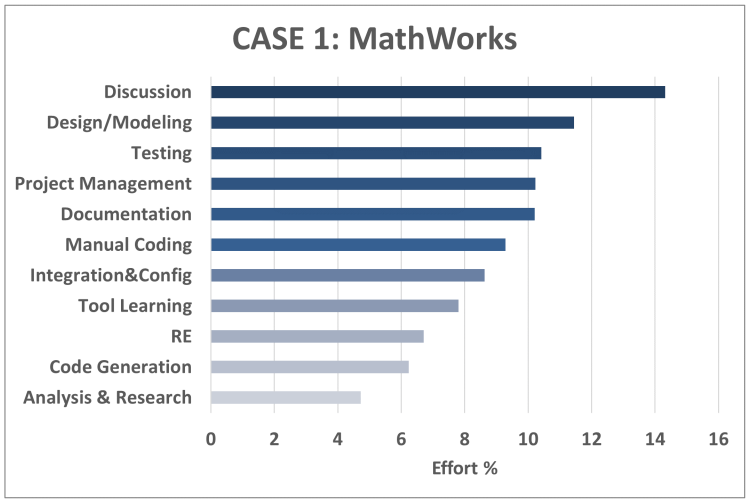


Figure 4.4: Total Effort Distribution Case 1

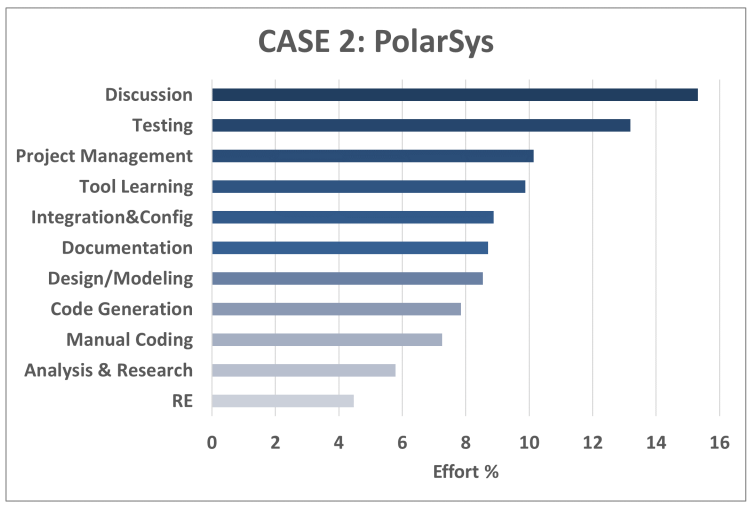


Figure 4.5: Total Effort Distribution Case 2

4.4.1 Development Efforts (R.Q.1)

Figures 4.4 and 4.5 orderly arrange the percentage values of the efforts spent on different MBE activities in case 1 (*MathWorks*) and case 2 (*PolarSys*), respectively. As can be noted, the majority of the effort is spent on *Discussion* (14.32% in case 1 (*MathWorks*) and 15.32% in case 2 (*PolarSys*)). More details

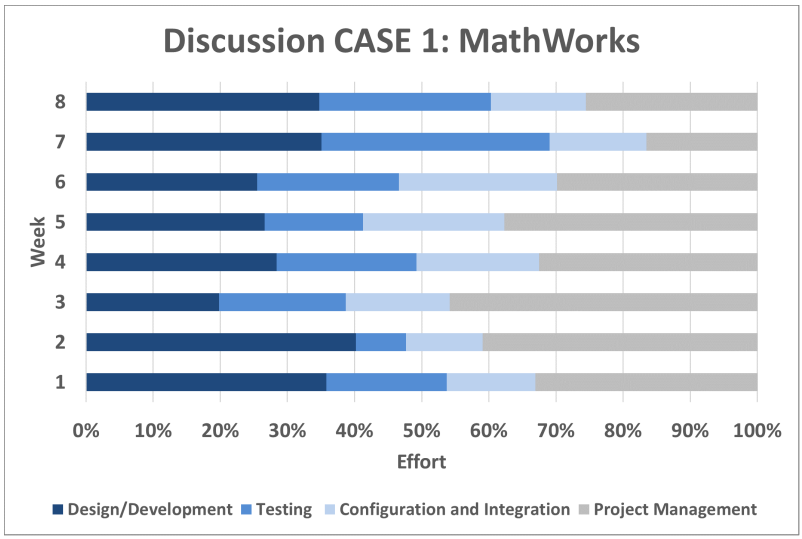


Figure 4.6: Discussion Effort Distribution Case 1

regarding the topics/arguments of the discussions related to case 1 and 2 are provided by Figures 4.6 and 4.7, respectively. In particular, these figures report an estimation of the percentages of the topics that were discussed each week. We got these estimations by matching the reported percentages on development discussions with the role responsibility of the reporting developer. Based on this, four main discussion topics were identified: Design and Development, Testing, Configuration and Integration, and Project Management. It can be observed that the majority of the discussions were about *Project Management* and *Design and Development*.

Table 4.1 shows a comparison of the effort phase distribution between the related work and our multiple-case study. This table is inspired by Papatheocharous [134]. First of all, it seems that the effort phase distributions in our two cases are compliant with the RUP's effort distribution reported by [127]. Moreover, it seems that the development effort distribution in the two cases is compliant with the "40-20-40" rule-of-thumb. This suggests that the effort distribution in MBE projects does not deviate from the distribution defined by the standard rules (*Proposition A*).

Giving a look on the separate development phases, we note the effort spent on *planning* (i.e., project management activities such as: define project scope, allocation, estimate cost, risks and schedule, etc.), *RE*, *specifications* and *testing* in our two cases seem to be in-line with the efforts reported by the related work, especially the recent works of Heijstek [136], Papatheocharous [134] and Yang [133]. Surprisingly, the effort spent on software design and modeling in

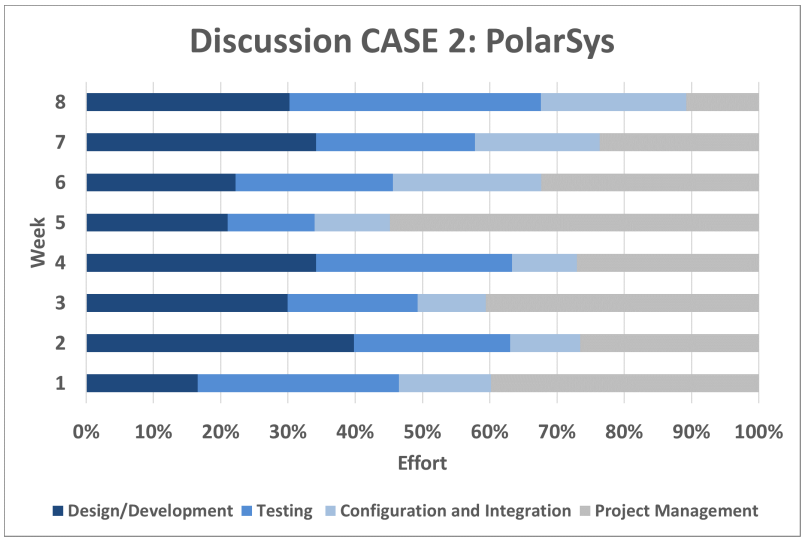


Figure 4.7: Discussion Effort Distribution Case 2

our two cases is similar to the findings of the other related work, especially [134], [133] and [131]. This finding empirically suggests that MBE approaches do not require a lot of effort on design and modeling, as it is *believed* among many developers. Moreover, code-generation based on models allowed our teams to spend less effort on manual coding. In particular, the combined effort spent on code-generation and manual coding together in each of our two cases is less that the effort of coding reported by [134, 136] and [133]. This empirical finding confirms that MBE approaches require less effort on manual coding because most of the code is obtained from models via code-generation.

By doing a cross-case analysis, we interestingly note that the effort distributions across phases of the two cases are quite similar to each other. The use of different modeling tools in the two cases, may explain the small differences in efforts spent on *Design and Modeling*. In particular, developers in case 1 *MathWorks* spent more effort on design and modeling than the developers of case 2 *PolarSys*. Hence, it might be that different modeling tools only have a small difference in impact on the development effort (*proposition C*).

Based on empirical findings, we suggest that MBE approaches *do not* require a lot of effort on design and modeling. Moreover, they require a *little* effort on manual coding, as most of the code is obtained from models via code-generation.

Table 4.1: Effort phase distribution reported in literature

Study	Planning	Requirements	Specifications	Design	Coding	Testing	Integration
Ambler (RUP) [127]		Inception (10%)	Elaboration (25%)		Construction (55%)		Transition (10%)
Zelkowitz [131]		RE (10%)	Spec (10%)	Design (15%)	Coding (20%)	Testing (45%)	
Brooks [130]	Planning (33%)				Coding (16%)	Testing (25%)	Integration (25%)
Sommerville [132]			Specs (15%)	Design (25%)	Development (20%)		Testing (40%)
Boehm [128]			RE - Analysis - Design (60%)		Coding (15%)		Testing (25%)
Pressman [126]			Analysis & Design (40%)		Coding (20%)		Testing & Integration (40%)
Papatheocharous [134]	Plan (9.6%)		Specs (9.3%)	Design (14.0%)	Build (42.3%)	Testing (18.2%)	Implement (6.6%)
Heijstek [136]	Planning (13%)	RE (8%)	Analysis & Design (11%)		Coding (38%)	Testing (12%)	Configuration (4%)
Yang [133]		Planning & RE (16.1%)		Design (14.9%)	Coding (40.3%)	Testing (21.6%)	Transition (7%)
Case Study 1 (MathWorks)	Planning (15.0%)	RE (10.0%)	Analysis & Research (7.0%)	Design & Modeling (17.0%)	Code Generation (9.2%) Manual Coding (13.6%)	Testing (15.4%)	Integration & Configuration (12.7%)
Case Study 2 (PolarSys)	Planning (15.3%)	RE (6.8%)	Analysis & Research (8.8%)	Design & Modeling (13.0%)	Code Generation (12.0%) Manual Coding (11.0%)	Testing (20.0%)	Integration & Configuration (13.2%)

4.4.2 Effort Distribution Over Time (R.Q.2)

Figures 4.8 and 4.9 present the effort distributions over each MBE activity during the two-month project period related to *MathWorks* case and *PolarSys* case, respectively. On the left side of the figures, eight main development activities are shown: Requirements Engineering, Analysis and Research, Design and Modeling, Code Generation, Manual Coding, Testing, Integration and Configuration, and Project Management. Whereas, on the right side of the figures, the effort distributions of other three secondary activities are reported: Documentation, Tool-Learning and Discussion.

By considering the effort spent on *design and modeling*, we notice a spike during the first week in both cases, *MathWorks* and *PolarSys*. This is because both teams used models at the beginning of the projects for ideation and discussion of design alternatives.

Team MathWorks spent more effort on *manual coding* than team PolarSys (as can be noticed by looking to figures 4.4 and 4.5), especially towards the end of the project. This phenomenon suggests that the code-generation facilities offered by MathWorks technologies are less than those of PolarSys. This is confirmed by the developers who reported that the tools offered by PolarSys (i.e., Papyrus and PapyrusRT) are more effective, user-friendly and generate code with more appropriate data structures and executables statements.

Both teams started with *testing* relatively early and throughout the projects. It is also notable that both teams spent more effort on *testing* towards the end of the projects. We think that this is a common trend in most software development projects, where more tests happen towards the end (e.g., system, integration and acceptance tests).

The effort spent on *integration and configuration* is quite similar between the two cases. Integration of software was occurring regularly in the two cases. In particular, for both cases, we notice a peak in the effort on week six. This is actually because the hardware was provided to both teams during that week, and the developers spent more effort on the configuration of the software and hardware.

As predicted, the effort on *tool learning* was high during the first weeks of the two cases, and gradually went down afterwards as the developers got more used to the tools over the time. The majority of the effort spent in the two cases was on *discussing* of the development activities. In particular, it seems that the discussions were regularly happening during the entire duration of the two projects, and not only during the planned weekly meetings.

Considering code-generation, we found that the tools offered by *PolarSys* open source technologies (i.e., Papyrus and Papyrus-RT) are *more mature* than the tools offered by *MathWorks* technologies (i.e., Matlab and Simulink).

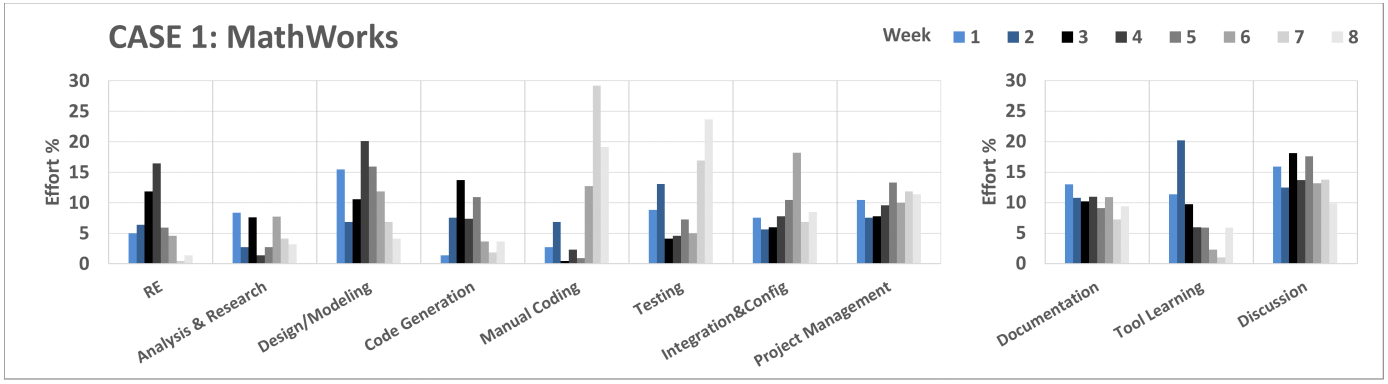


Figure 4.8: Effort Distribution Case 1

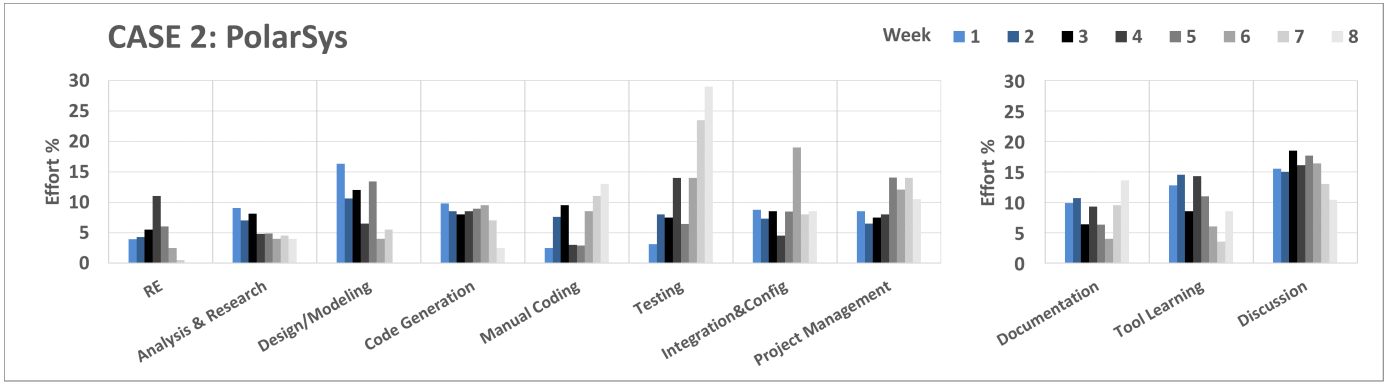


Figure 4.9: Effort Distribution Case 2

4.4.3 Individual vs. Collaborative Effort (R.Q.3)

In the weekly questionnaire, we asked developers about their perceptions of the ratio of individual versus collaborative development effort that occurred during each week. Figures 4.10 and 4.11 provide an overview of the distribution of collaborative work over the entire duration of the project of case 1 *MathWorks* and case 2 *PolarSys*, respectively.

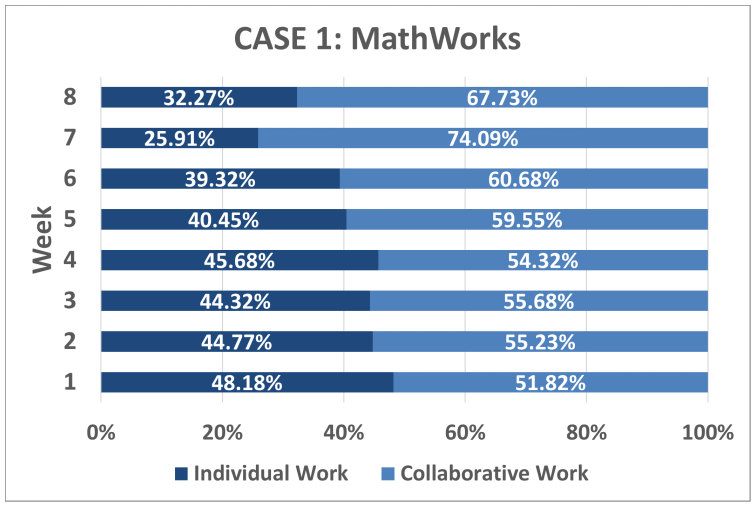


Figure 4.10: Case 1: individual versus collaborative work

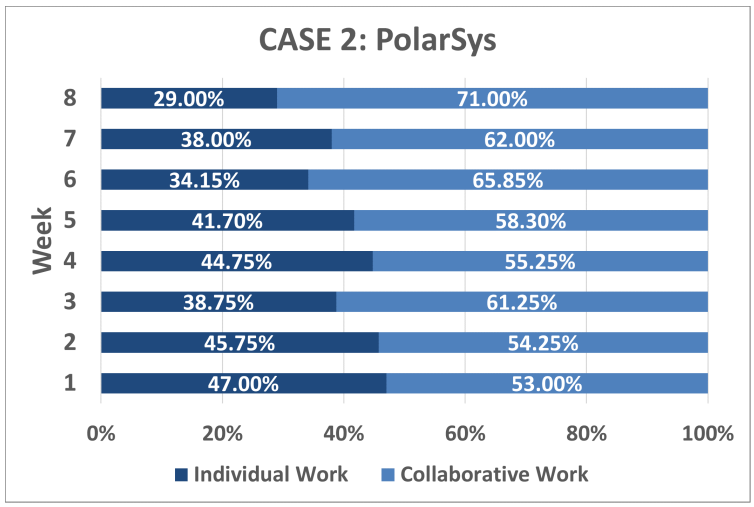


Figure 4.11: Case 2: individual versus collaborative work

Apparently in both projects, and for each week, the collaborative work was dominating. The collaborative work included discussions, group meetings, sharing knowledge and understanding, and collaborative development (e.g., definition of the data structure of the maze, modeling, code-generation, testing and pair programming). A further analysis of the patterns in figures 4.10 and 4.11 shows that more individual work happened at the beginning of both development projects, while more collaborative work happened towards the end. This can be explained by the fact that the developers worked individually on tools exploration and learning during the first weeks. Moreover, the developers reported that they spent more time on pair programming and testing meetings towards the end of the projects.

Our findings empirically indicate that model-based software development is an endeavor that requires intensive *communication and collaboration* between developers.

4.4.4 Tool-Chain

A variety of tools were used for the different development activities. These tools ranged from being main model-based development tools, such as Papyrus, PapyrusRT, Matlab/Simulink, Enterprise Architect and Capella, to other supportive tools such as Latex, Slack, PDF Reader, Version Control, Text Editor, Outlook, Power Point and Word. Figure 4.12 and 4.13 provide an overview on the tools which were used in the two cases: 1 and 2. These figures also highlight the distribution of the total tool effort percentage between the different used tools. About 22% of case 1 total tool-use effort was spent on Matlab and Simulink. Whereas, around 30% of case 2 total tool-use effort was spent on Papyrus and Papyrus RT. The focus on these tools was expected given the project's objective of tool use of the two development teams.

4.4.5 Experienced Challenges (R.Q.4)

Every week, the members of each project were asked to report the challenges that they experienced during the past week. The pie charts 4.14 and 4.15 orderly arrange the reported challenges ranging from the most experienced to the less experienced challenge during the execution of case 1 and case 2, respectively. *Tools Usability* was the most experienced challenge to MBE in the two cases (23% in case 1 and 25% in case 2). Challenges in *Tool-Chain Learning* were the second most experienced challenges (20% in case 1 and 19% in case 2). More details regarding the categories of the challenges are described in Table 4.2.

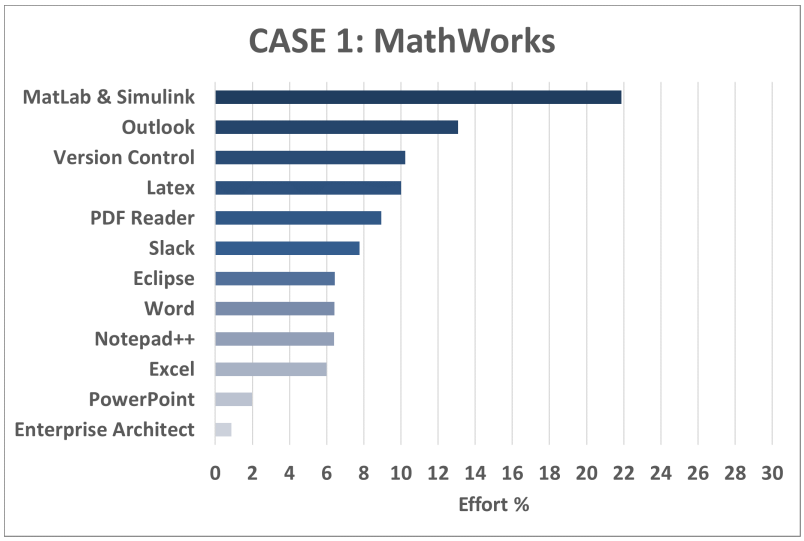


Figure 4.12: Case 1: Overview of the used tools and their effort distribution

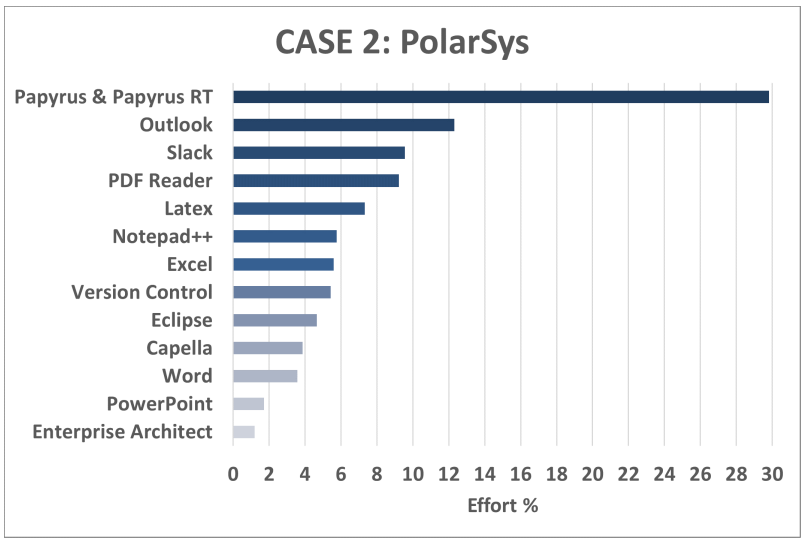


Figure 4.13: Case 2: Overview of the used tools and their effort distribution

In both of the cases in our study, multiple challenges related to MBE tools were reported, such as tools- usability, learning, installation and configuration, and update. This is actually in-line with the related work (e.g., [22] and [14])

which states that poor tool-support is one of the main challenges to MBE (*proposition B*).

A cross-case analysis shows that the majority of the challenges were overall experienced similarly in both cases, especially tool-related challenges (*proposition D*). This finding indicates that the modeling tools provided by *MathWorks* and *PolarSys* are still immature and have to be enhanced in order to meet the needs of MBE and MBE developers. More information on *MathWorks* and *PolarSys* challenges are provided on-line[11].

Our findings show that tool-related challenges are the most encountered. These tool-challenges are due to: tools usability, toolchain learning, interoperability of tools, and tools installation and configuration.

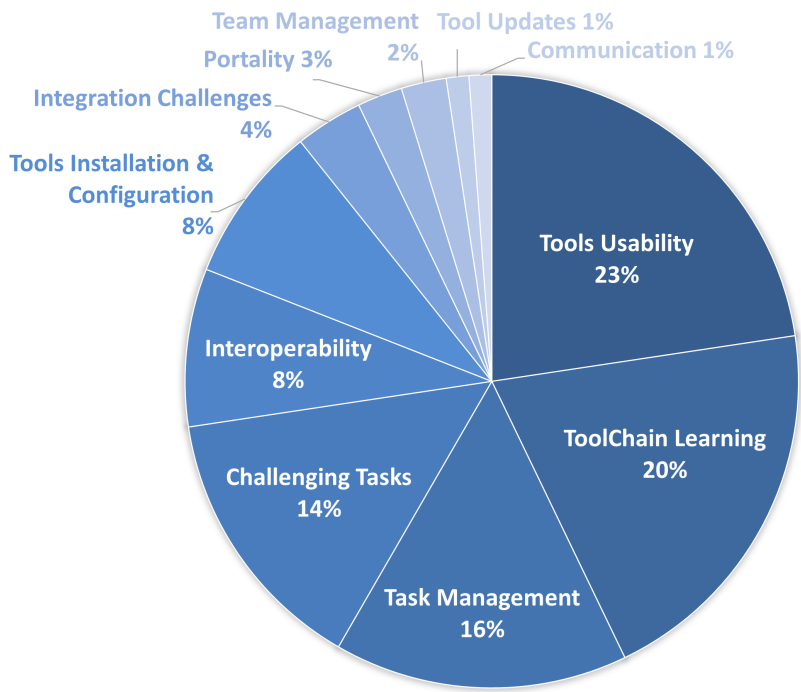


Figure 4.14: Experienced challenges in Case 1

[11] http://www.rodijolak.com/pdf/toolsChallenges.pdf

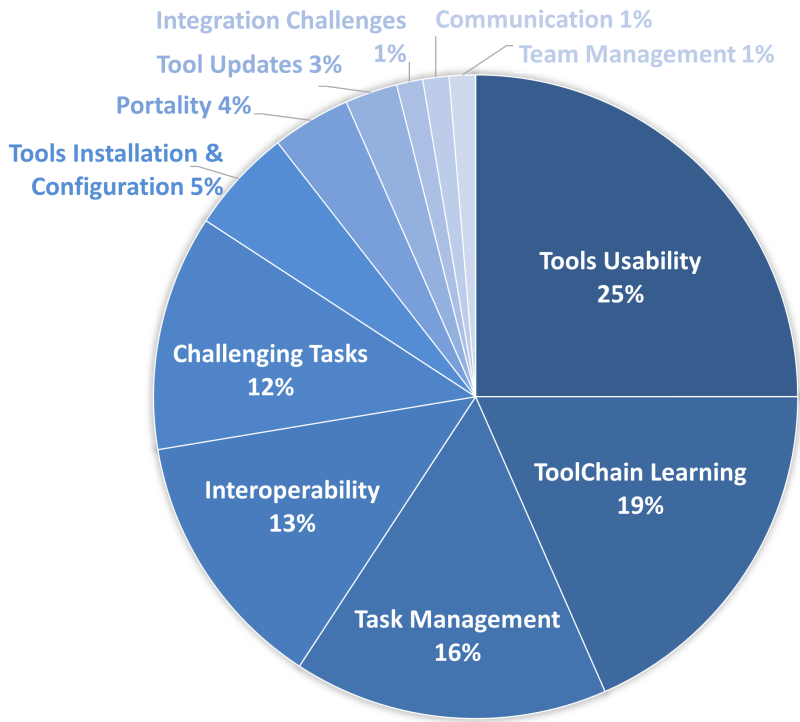


Figure 4.15: Experienced challenges in Case 2.

Table 4.2: Classification schema for the challenges

Category		Description
Tool-Chain Learning		Effort on learning the tools to be used in the project
Tools Installation & Configuration (I&C)		Effort on the I&C of the tools on the machines of the developers
Interoperability		Missing the ability to exchange artifacts between different tools
Tools Update		Effort on adapting the software to new tool/library versions
Tools Usability	Difficulty of Use	Complexity and cumbersomeness of the tools
	Effectiveness	Incompleteness, inaccuracy and inconsistency
	Efficiency	Long tasks' completion time
	UX	Uncomfortable tools and unacceptability of use
Task Management	Task Allocation	Effort on organization & distribution of tasks between developers
	Synchronization	Effort on synchronization of development activities
Team Management		Effort on organization of the teams
Challenging Tasks		Complex development tasks that require a lot of mental effort
Portality		Difficulty in transferring software on different platforms (i.e., operative systems)
Integration Challenges		Difficulty in integrating single different modules of the software
Communication		Problems caused by miss- or late communication between the stakeholders

4.4.6 Challenges Distribution Over Time (R.Q.5)

Figure 4.16 presents the distribution of the experienced challenges over the eight weeks period of the two projects. It is remarkable that *Tool-chain Learning* and *Tools Usability* challenges were mostly experienced and reported during the first two weeks of the two project.

Challenges in tool-learning were also perceived in weeks 3 and 4 in both cases. For case 1 (*MathWorks*), the main reason was that the developers spent a lot of time on learning Simulink testing-tool in order to test the simulator software. Whereas for case 2 (*PolarSys*), learning how to use third-party C++ code in a PapyrusRT project was perceived as challenging.

Tools usability challenges were reported regularly during the execution of the two projects. In particular, on week 6 challenges related to Simulink model advisor were reported in case *MathWorks*. In contrast, on weeks 5 and 6 several challenges were reported by team *PolarSys* related to a PapyrusRT update from version 0.7 to 0.8, which in turn caused lots of migration conflict issues.

Generally, it seems that the majority of the issues related to tools usability were encountered during the first weeks, when the developers started to get their hands dirty in the projects. More tool-related issues were reported afterwards by performing more activities in the projects, and hence by exploring and using more tools' functionalities.

For both cases, *Task management* challenges were basically encountered at the beginning, when the teams spent more effort on distributing the tasks between the developers, and also in different occasions afterwards (until week 7) where synchronization issues were reported.

In both cases *Challenging tasks* (see Table 4.2) were more encountered at the beginning- and towards the end of the projects. At the beginning, performing development activities was challenging as the developers were not so familiar with the tools. Whilst in order to finish the projects on time, the developers were over-allocated with multiple tasks towards the end of the projects.

Tools installation and configuration challenges were encountered during the first weeks of the two projects, as could be expected.

For both cases, tool-*interoperability* challenges were mainly encountered from week 3 to week 6 when several issues related to linking the produced software application with the API of the rover (e.g., coding the wrapper between the generated code and rover's API) were reported. This also caused *Portability* challenges as there were incompatibilities between some API-library dependencies and the used operative systems.

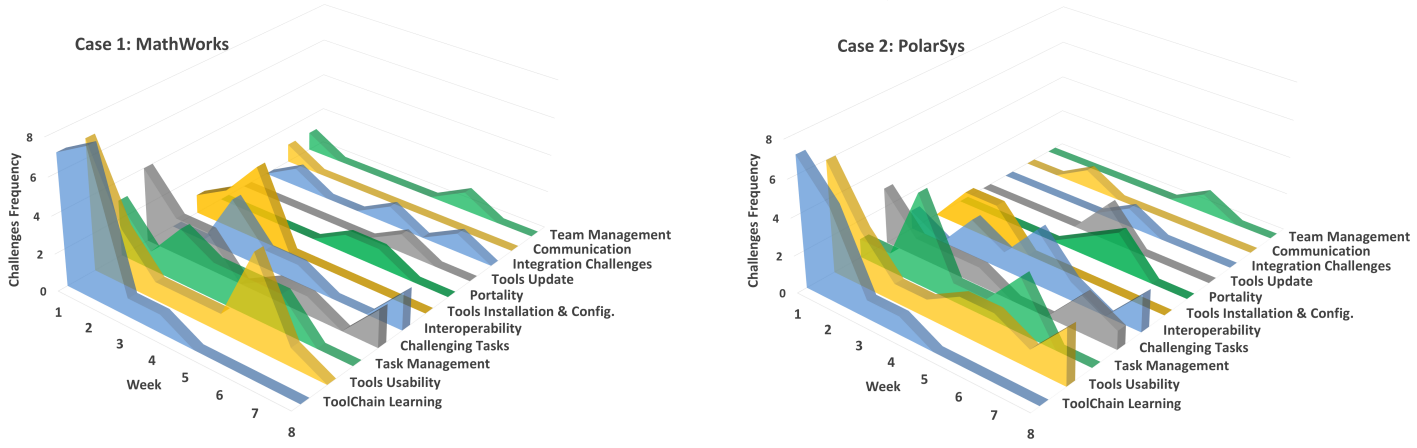


Figure 4.16: The distribution of the experienced challenges over the total period of the project.

4.5 Threats to Validity

We identified and grouped the threats to validity in our study according to Yin [78]:

4.5.1 Construct Validity

Constructs validity refers to how well operational measures represent what researchers intended them to represent in the study in question. The collection of subjective perceptions regarding development efforts and challenges after completing a project may not be optimal. This is because the subjects may fail to recall how much effort was given to a specific task or what challenges were encountered during their experience. To mitigate this, we collected the perceptions on a weekly basis: Once after the end of each week. Furthermore, we looked into the logs of the modeling tools and *Procrasti* activity tracker in order to triangulate the data which we got through collecting perceptions. In turn, the activity tracking and logging tools have a limitation. These tools log the activities only when there is an interaction between the users and their PCs. As a result, no activity does not imply that the subject is not working, for example one might be reading the document without touching the PC. We think that the two data collection approaches (i.e., collecting perceptions and logging developers' activities) adopted in this study have their own limitations. However, using multiple sources of evidence helped us to increase construct validity by encouraging convergent lines of inquiry.

4.5.2 Internal Validity

Internal validity concerns studies in which causal relationships are examined. Moreover, it concerns efforts made to ensure that possible confounding factors are identified and alleviated. The level of experience and expertise in MBE may influence the effort required by a developer to accomplish a specific MBE activity. This may lead to spending more or less effort on the development tasks. All of the subjects who took a part in our study are familiar with MBE because they participated in a workshop that taught the MBE development paradigm.

Some developers may consider some development tasks as challenging, while other developers may consider such tasks as less challenging. The subjects often discussed their reported challenges and motivated their perceptions. This was, to some extent, helpful to conceive the seriousness of the reported challenges. Moreover, we consider the challenges that were encountered and reported by more than one subject as more significant.

We recall that our subjects are Professional Doctorate in Engineering (PDEng) trainees. This might have made the subject spending more effort on learning new tools and finding out how to work as an actual development team.

However, our subjects know each other in advance. Furthermore, prior to our study, the subjects worked together on several other development projects.

4.5.3 External Validity

External validity concerns the extent to which results of a case study can be generalized. By design, case studies have a very limited external validity stemming from the fact that a topic is studied within its context. Therefore, we cannot claim that our findings are generalizable (i.e, generalizations to different projects in different domains might have different results). Instead, the case design and the replication logic with the cross-case analysis increases the external validity of this study. In particular, we tried to describe the case context as detailed as possible in order to allow practitioners to decide whether or not the findings might generalize to their own case context. Moreover, we underline that our study involved first-time tool users. Therefore, different results might be obtained if professionals with deep tool experience did the same projects.

4.5.4 Reliability

Reliability concerns the extent to which the operations of a study can be repeated by other researchers, achieving the same results. As a part of the case study design, we created a case study protocol which ensured that we conducted the study and collected the data in a consistent manner. By using this protocol, we believe that the study can be reproduced by other researchers.

4.6 Conclusion and Future Work

In this paper we studied the effort distribution across various tasks for two projects that use different MBE tool-chains for developing an autonomous MARS rover. We obtained data both from the automatic logging of the tool-activities on the developers' computers as well as via weekly questionnaires.

Our study showed the patterns of effort distribution in MBE across different development activities as well as over time. This shows that there is no penalty in building models as part of the construction phase. Our study is the first to show that collaborative tasks make up the major part of the total of all development tasks. The resulting observations on effort distribution of this study could lead to improved MBE project planning and organization, which in turn could lead to cost reduction.

Our inquiry into challenges showed that tool-related challenges are the most encountered. We uncover that specific tool-challenges are due to: i) usability of the tools, ii) the learning of the tool-chain, iii) the interoperability of various tools and iv) the installation and configuration of the tools. Exposing such

challenges would make them a candidate subject for research that are concerned with MBE process improvement. Moreover, understanding and providing ways to overcome these challenges could bring a significant impact to the effectiveness and efficiency of the MBE approach.

4.6.1 Future Work

As we found that the majority of the development effort is spent on the *collaboration and communication* activities, we would like to explore the effect of using models on software design communication. This in order to understand whether or not the use and share of software models could help in communicating and discussing software architectural/design decisions.

Acknowledgement

The authors would like to thank Engr. Nontas Rontogiannis for his review and constructive feedback on this work.

Chapter 5

OctoBubbles

(D) R. Jolak, K.D. Le, K.B. Sener, M.R.V. Chaudron "OctoBubbles: A Multi-view Interactive Environment for Concurrent Visualization and Synchronization of UML Models and Code"
In the 25th IEEE International Conference on Software Analysis, Evolution and Re-engineering (SANER), pp. 482-486. 2018.

Abstract

The process of software understanding often requires developers to consult both high- and low-level software artifacts (i.e. models and code). The creation and persistence of such artifacts often take place in different environments, as well as seldom in one single environment. In both cases, software models and code fragments are viewable separately making the workspace overcrowded with many opened interfaces and tabs. In such a situation, developers might lose the big picture and spend unnecessary effort on navigation and locating the artifact of interest. To assist program comprehension and tackle the problem of software navigation, we present *OctoBubbles*, a multi-view interactive environment for concurrent visualization and synchronization of software models and code. A preliminary evaluation of *OctoBubbles* with 15 professional developers shows a high level of interest, and points out to potential benefits. Furthermore, we present a future plan to quantitatively investigate the effectiveness of the environment.

Keywords: Model-Based Software Engineering; Modeling; Code; Multiple Views; Synchronization; Design Environment; UML.

5.1 Introduction

Model-based approaches aim to raise the abstraction level in system specification by allowing developers to think in terms of conceptual ideas rather than in terms of their details [141]. Software designers create models in order to express their ideas and document their decisions regarding a software system. Successively, the created models are used to communicate the design decisions to other stakeholders, e.g. developers. In turn, developers use these models, which are considered as instructions (*blueprints*) for system construction [3], to implement the software system. Before and during the implementation phase, the developers seek to understand both the structural and behavioral aspects of the system. The process of understanding often requires developers to create a mental representation of high-level descriptive domain artifacts (i.e. models) and low-level artifacts (i.e. code) [42, 43].

The Unified Modeling Language (UML) provides a standard way to visualize, specify, construct and document software systems [142]. Most of UML Computer-Aided Software Engineering (CASE) tools like ArgoUML[1], Visual Paradigm[2] and ObjectAid[3] support forward, reverse and round-trip engineering [89]. However, these tools do not provide developers with multiple simultaneously-visible views of both low- and high-level software artifacts. The developers who use these tools are constrained to open views on different software artifacts making the workspace overcrowded with many opened interfaces and tabs. In such a situation, the developers often lose the big picture and spend unnecessary effort on navigation and locating the artifact of interest. It is reported that *at most* 35% of developers time is spent on software navigation [44], and *around* 60% of developers time is given to software understanding activities [45]. Moreover, the use of concurrent and multi-view interfaces is proposed for tasks in which a mental effort is required to perform a comparison between the different parts of complex systems [143].

In this paper, we present *OctoBubbles* to assist program comprehension and tackle the problem of software navigation. *OctoBubbles* is a multi-view interactive environment for concurrent visualization and synchronization of both high- and low-level software artifacts (i.e. models and code).

The main contributions of this paper are two-fold. First, a design of a novel *scaling* approach which is used to provide an interactive, bidirectional, and smooth navigation between models and code. Second, the results of a qualitative user evaluation of *OctoBubbles* indicating a high level of interest, and pointing out to potential benefits and future improvements. Furthermore, in this paper we present a *future* plan for a quantitative evaluation of the effectiveness of *OctoBubbles*.

[1] `http://argouml.tigris.org/`
[2] `https://www.visual-paradigm.com/`
[3] `http://www.objectaid.com/`

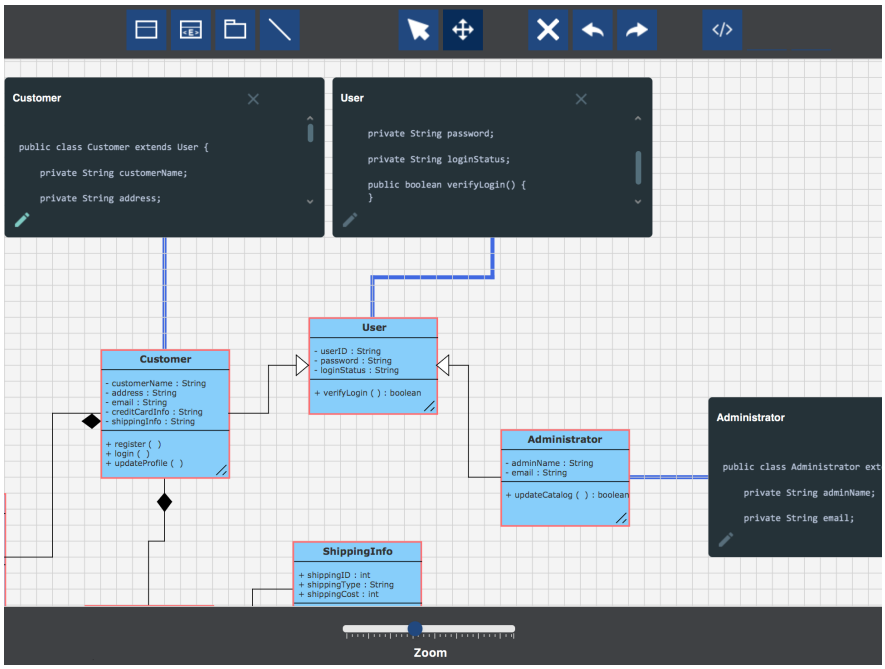


Figure 5.1: A part of the main canvas of OctoBubbles. The buttons at the top are (*from left to right*): create class, create enumeration class, create package, create association, select, pan, delete, undo, redo and visualize code.

5.2 Approach

OctoBubbles aims to facilitate software comprehension by bridging the gap between high- and low-level software artifacts. Also, it aims to reduce the cognitive load placed on developers by getting rid of disorienting view switches between models and code. *OctoBubbles* is an open source software development environment based on *OctoUML* project [8,83,117]. It supports multiple modes of interaction e.g. mouse, keyboard and touch. It can be deployed on a variety of devices ranging from PCs over touch screens to large interactive whiteboards. Figure 5.1 shows the main interface of *OctoBubbles*. It provides an overview on how the UML model and the corresponding source code are concurrently visualized on the same canvas.

The environment is designed to support and improve the synchronization and visualization process of models and code.

5.2.1 The Synchronization Mechanism

The synchronization mechanism of *OctoBubbles* is responsible of transmitting changes in the UML model to the source code and vice versa. This mechanism is based on the *JavaParser* library. This library provides means to convert the source code into a Abstract Syntax Tree (AST) structure. The AST structure is used for matching the altered artifact with its corresponding low- or high-level artifact. By this mechanism, the users can modify the details of one of the two artifacts (UML model or source code), and *OctoBubbles* keeps the other one synchronized. This means any modification in the source code committed by the user will automatically propagate to the class model and vice versa.

5.2.2 The Visualization Mechanism

In contrast, the visualization mechanism of *OctoBubbles* is based on a novel *scaling* approach. To have an idea on how this approach does support the concurrent visualization of models and code, let us consider the following scenario:

Bob draws a UML class model as in Figure 5.2 (A). He then selects one or *more* classes (in this case ClassD is selected), and clicks the </> button (shown in the top of Figure 5.1). Doing this, *OctoBubbles* either matches the selected class with the existing relative source code or generates a source code skeleton from scratch according to the predefined attributes and operations. *OctoBubbles* also draws an invisible borderline which surrounds the model and splits it into four areas: top-left, top-right, bottom-right, and bottom-left. Moreover, it detects in which area the majority of the selected class resides (in this case the top-left area), as it is shown in Figure 5.2 (B).

After detecting the area of interest, *OctoBubbles* further splits the external space which is surrounding the UML class model into eight external regions: (i) four main regions: top, right, bottom and left, and (ii) four reserve regions: top-left, top-right, bottom-right and bottom-left (see Figure 5.3 (C)). These regions are used for code visualization. *OctoBubbles* first seeks to visualize the code in the main regions, however; in case of lack of space in the main regions, the reserve regions are then used for code visualization.

There are three external regions which are adjacent to the area of interest (see Figure 5.3 (C)). Two are main regions (left and top), and one is a reserve region (top-left). These regions become candidates for visualizing the source code of classD. As it can be observed from Figure 5.3 (C), there is an empty space in both of the two main external regions. To decide in which region the source code should be visualized (displayed), *OctoBubbles* calculates the distances from the sides of the selected class box (i.e. ClassD box) to the opposite borderlines between the area of interest and the external main regions (in this case α and β, see Figure 5.3 (D)).

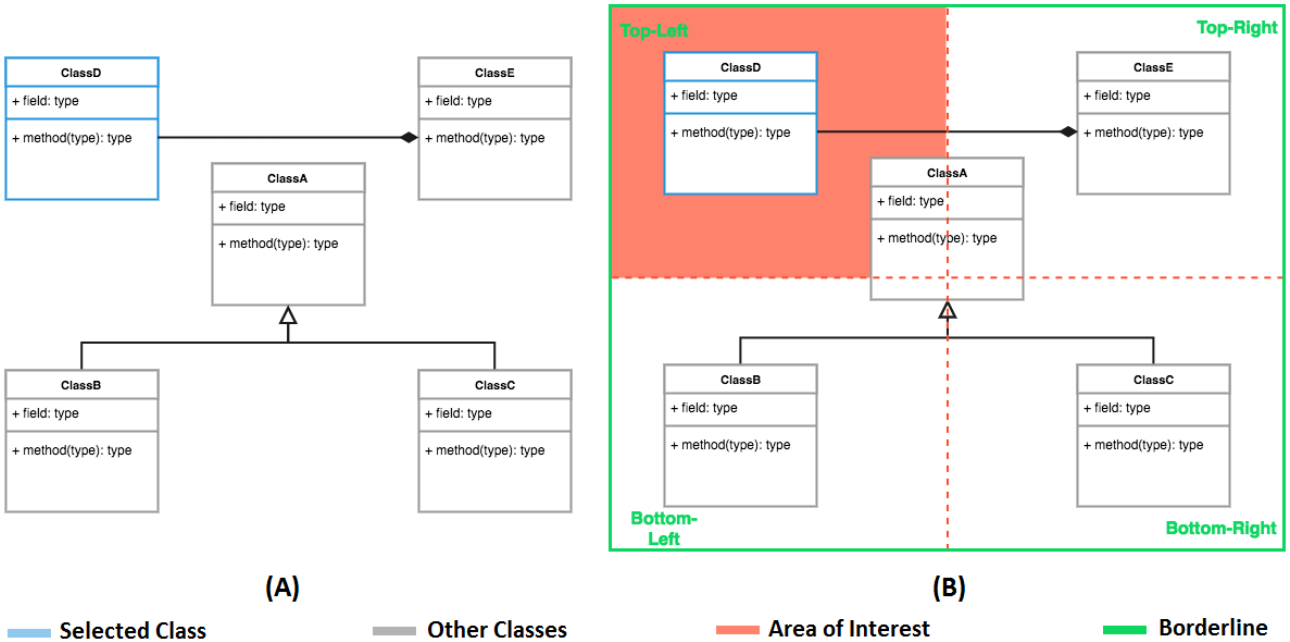


Figure 5.2: The scaling approach: steps A and B

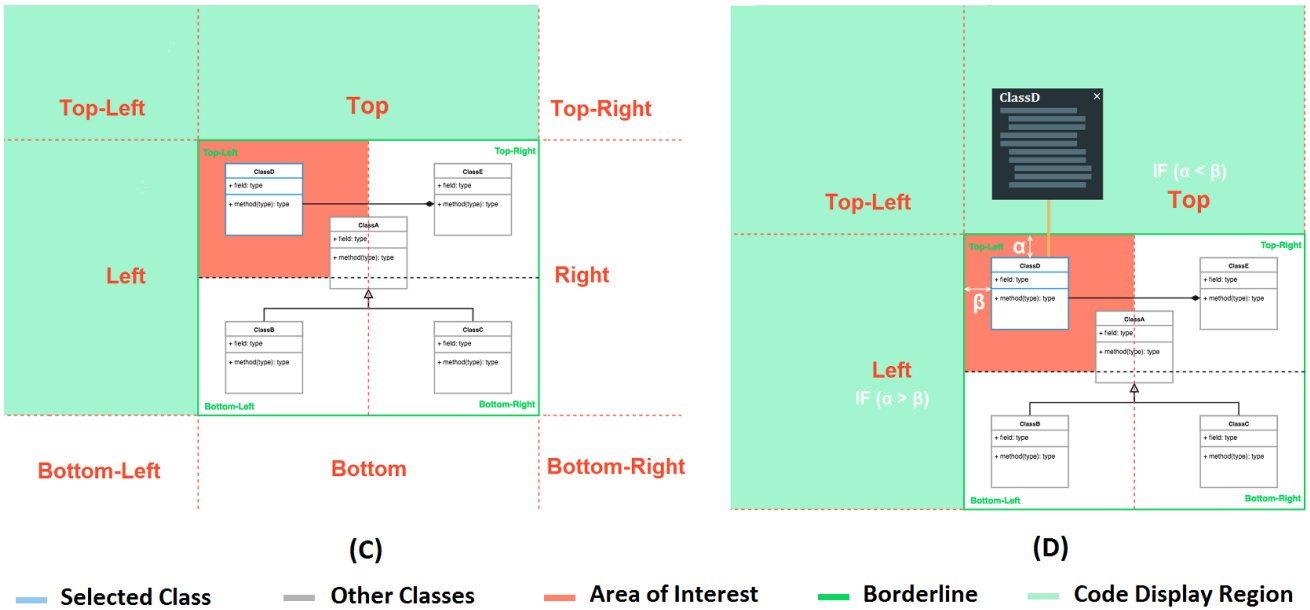


Figure 5.3: The scaling approach: steps C and D

Finally, *OctoBubbles* chooses the closest external main region to the selected class and visualizes the source code there (in this case the *top* region, because $\alpha < \beta$).

In case that $\alpha = \beta$, then *OctoBubbles* chooses the region which contains less visualized source code bubbles. Moreover, in case of lack of space in the candidate external main region, *OctoBubbles* visualizes the source code bubble in the closest external reserve region to the area of interest (in this case the *top-left* region).

The users of *OctoBubbles* can arbitrarily close and change the location of any source code bubble and any class box. When the users change the location or size of a class box, the location of its corresponding visualized code bubble can be either updated automatically or manually by: (i) closing the code bubble, (ii) selecting the target class, and (iii) clicking the </> button.

Figure 5.4 summarizes the overall synchronization and visualization process of *OctoBubbles*. Furthermore, to get a better idea on how *OctoBubbles* works, check out the demo video[4].

5.3 Preliminary Evaluation

For a preliminary evaluation of *OctoBubbles*, we formed the following research questions:

- R.Q.1 What are users' perceptions regarding the idea, usability and efficiency of *OctoBubbles*?
- R.Q.2 What are the missing and desired functionalities?

In order to answer these questions, we conducted a user study. Fifteen subjects were involved in this study. All subjects are working in industry as software developers. All subjects have knowledge in UML and object oriented programming. The user study included three main tasks:

- **Task I** to perform a round-trip engineering assignment[5]. The assignment was to create a UML class diagram of a given scenario. Moreover, to perform synchronization and modification of both the created UML class diagram and the associated source code,
- **Task II** to answer ten closed questions regarding the usability of *OctoBubbles*, and
- **Task III** to answer five open questions on the perceived usefulness (1 *question* (q)) and efficiency of *OctoBubbles* ($2q$s), missing functionalities ($1q$) and desired features ($1q$) for future development of the system.

[4] `http://rodijolak.com/#octobubbles`
[5] `http://rodijolak.com/pdf/OctoBubbles_Assignment.pdf`

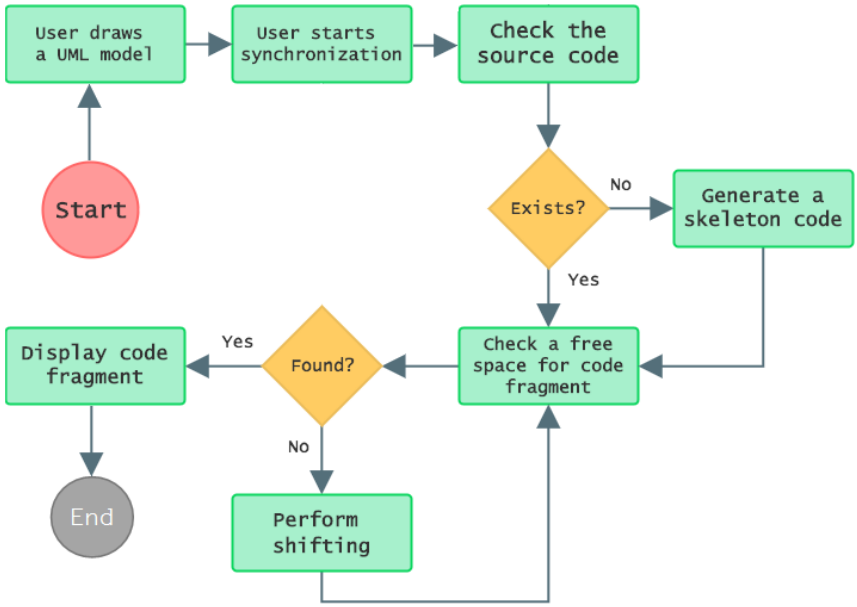


Figure 5.4: The synchronization and visualization process of *OctoBubbles*

Having done Task I, the subjects were asked to answer ten closed questions regarding the usability (*Task II*). Table 5.1 presents the obtained results. For each usability measurement, the subjects gave a score ranging form 1 (the lowest score) to 5 (the highest score). *Med* is the median score, *Q1* indicates the first quartile, *Q3* is the third quartile, and *IQR* stands for Inter-Quartile Range (i.e. *Q3*-*Q1*).

We qualitatively analyzed the answers to the five open questions (*Task III*). In particular, we present the answers of the subjects according to the following three themes:

5.3.0.1 Potential Benefits of OctoBubbles

The subjects stated that *OctoBubbles* helps to better establish traceability between a UML model and its associated source code, especially in software testing and maintenance tasks. For such tasks, developers might jump back and forth between low- and high-level artifacts to have a better understanding of the system. The visualization approach of *OctoBubbles* was considered extremely useful to have an abstract view of a software system, concurrently with a more detailed view of its aspects.

Table 5.1: Perceptions regarding the usability of OctoBubbles

Usability Measurement	Med	Q1	Q3	IQR
Willing to use the system frequently	4	3	4	1
Complexity of the system	1	1	1	0
Ease of use	5	4	5	1
Need of support to use the system	1	1	1	0
Integrity of various functions	5	4	5	1
Inconsistency in the system	1	1	1	0
Intuitiveness	4	4	5	1
Cumbersomeness to use	1	1	2	1
Feeling confident when using the system	4	4	5	1
Required learning-effort	1	1	2	1

5.3.0.2 Perceived Efficiency

According to our subjects' experience, *OctoBubbles* was perceived more efficient than other CASE tools that support forward, reverse and round-trip engineering. One of the subjects uses *StarUML* (http://staruml.io). Comparing *StarUML* with our approach, the subject stated that *OctoBubbles* is more efficient and user friendly. The synchronization mechanism of *OctoBubbles* was perceived pretty smooth and extremely fast. Moreover, the visualization mechanism of *OctoBubbles* was considered very helpful in saving the effort of artifacts navigation.

5.3.0.3 Missing Functionalities & Desired Features

There was a demand to enable *OctoBubbles* of supporting a concurrent visualization of software behavioral models (e.g. *Sequence Diagrams*) and their associated source code. The motivation behind this demand is that high- and low-level behavioural artifacts would complement the role of the structural artifacts in describing the software system. Indeed, the behavioral artifacts describe the software system from a different perspective, and this is considered helpful to better support program comprehension. The *find-method-usage* feature in most integrated development environments was suggested to be supported by *OctoBubbles*. This in order to show methods' usage-links between the visualized code fragments, but also between the elements of the model. Finally, the subjects suggested to make use of the voice interaction modality of *OctoUML* [144] to support the visualization mechanism of *OctoBubbles*.

5.4 Related Work

There are several UML CASE tools that allow the transformation of UML models into source code and vice versa. Examples are Visual paradigm, Ar-

goUML, Altova UModel[6], yWorks UML Doclet[7] and ObjectAid. These tools support the creation of several types of diagrams like UML, entity-relationship and business process modeling diagrams, as well as various programming languages such as Java, C++ and C#. None of these tools allow for a concurrent visualization, as well as modification, of models and multiple code fragments on the same interface. Therefore, these tools may actually increase the cognitive load placed on users by forcing disorienting view switches between different software artifacts.

Lethbridge et al. [145] created *Umple*, a software development technology that merges modeling and programming. *Umple* aims to enhance program understanding by letting developers to work at the abstract level, and concurrently see diagrammatic and *Umple*'s textual representations of the source code. Once the diagram is created, the users of *Umple* can generate source code in different languages such as, Java, Ruby and PHP. Compared to *Umple*, *OctoBubbles* does not require the users to learn a specific textual language to create diagrams. Indeed, *OctoBubbles* allows a *direct* bi-directional mapping between the model and the source code (for example, Java). Furthermore the visualization approaches of *Umple* and *OctoBubbles* are different. *OctoBubbles* aims to reduce the navigation and traceability efforts by visualizing code fragments as close as possible to their corresponding class models. Moreover, the low- and high- level software artifacts in *OctoBubbles* are visualized concurrently on the same canvas (one screen), and the users can zoom-in/out to the artifact of interest. In contrast, *Umple* visualizes the diagram and the *Umple*'s textual representation in two different juxtaposed scrollable screens, and the users have to scroll-up/down in order to locate the artifact of interest.

Bragdon et al. [146] proposed a working set-based interface to support software development and ease code understanding and maintenance. Their interface allows a concurrent visualization of multiple lightweight and editable code fragments called *Code Bubbles*. The evaluations showed that their approach helps in improving code understanding time and decreasing navigation interactions. *Code bubbles* is concerned with source code visualization and management. In contrast, *OctoBubbles* allows the concurrent visualization of models and source code, as well as supports the management and the synchronization of these two artifacts.

Baltes et al. [75] stated that informal sketches and diagrams are often detached from the source code they document. The authors conducted a study to understand the use of sketches in software engineering. They found that sketches are considered helpful to understand the related source code artifacts. For this, the authors proposed *SketchLink* to let software developers easily capture, annotate, and link their diagrams and sketches to the correspondent source code artifacts. In contrast to *OctoBubbles*, *SketchLink* provides a

[6] `https://www.altova.com/umodel/`

[7] `https://www.yworks.com/products/ydoc`

matching between early-phase sketches and the corresponding source code. Furthermore, *SketchLink* does not support a bi-directional synchronization of the two artifacts, together with the links in between when they are updated or deleted.

5.5 Conclusion and Future Evaluation Plan

While performing software comprehension and maintenance tasks, software developers often use different interfaces to navigate complex and often implicit relationships between software artifatcs [147]. This actually requires developers to spend an extra effort to locate the artifacts of interest. In this paper we presented *OctoBubbles*, a multi-view interactive environment for concurrent visualization and synchronization of UML models and code. *OctoBubbles* aims to support program comprehension, facilitate traceability and tackle the problem of software navigation by avoiding disorienting view switches between high- and low-level software artifacts.

In order to study the effectiveness of *OctoBubbles*, we aim to investigate achievable improvements in developers' daily practice using *OctoBubbles*. We also aim to understand whether *OctoBubbles* can reduce developers' workload and mental demand when they have to switch their focus between UML class diagrams and the related source code. The task of the study will require the participants to understand and evolve an object-oriented-programming based system. This task is frequently performed in software development processes such as software maintenance, where combining models and source code has been claimed to improve the efficacy of the process [148, 149]. The quantitative measures that would be used in the study are both objective and subjective.

Objective Measures. Task completion time and the number of committed mistakes during the execution of the task will be used to measure participants' performance. We also plan to use quantitative measures of eye gaze such as pupil's size and saccades, as they have been used to measure cognitive workload [150, 151] and to evaluate user interfaces [152].

Subjective Measures. *NASA TLX* and *SUS* will be used to examine perceived cognitive workload and analyze users' subjective preferences on the usability of *OctoBubbles*. These measures have been widely used in prior works to evaluate the mental impacts of CASE tools on software developers [153].

The planned user study should adequately examine the effectiveness of *OctoBubbles*. We also believe that the user study can be applied to evaluate similar systems in the future.

Chapter 6

Does Distance Still Matter? Revisiting Collaborative Distributed Software Design

(E) R. Jolak, A. Wortmann, M.R.V. Chaudron, B. Rumpe "Does Distance Still Matter? Revisiting Collaborative Distributed Software Design" *In IEEE Software 35, no. 6: 40-47. 2018.*

Abstract

Global software engineering requires supporting distributed collaboration for most software development activities. However, geographical distance challenges effective collaboration. Nowadays, we are witnessing significant advances in communication and collaboration technologies. So, we explored whether these advances enable effective remote collaboration. To that end, we studied the design activities of both colocated and distributed professional software designers. The findings are based on analysis of video recordings of design sessions and questionnaires. We found that despite comprehensive technological improvements, distance still matters. To ensure effective distributed software design, designers must consider extra (nontechnical) details.

Keywords: Software Engineering; Collaborative Design; Communication; Distance; GSE

6.1 Introduction

Companies engage in global software engineering (GSE) to reduce development time and costs. Companies also head toward cross-site distribution of their development work to take advantage of proximity to markets and customers [25]. However, working at a distance might compromise the effectiveness of GSE. There are two important challenges to making GSE successful. Almost two decades ago, Gary Olson and Judith Olson raised these challenges [154]:

- *technological challenges* raised by the need for efficient, effective remote-collaboration tools and media; and
- *social challenges* raised by differences in local context, culture, language, and trust between collaborators.

They predicted that future technological advances will reduce the effect of the technological challenges. But they also predicted that working at a distance will rarely succeed owing to the inevitable differences raised by the social challenges. However, advances in communication and collaboration technologies raise the question of whether distance still matters.

One of the key activities of software engineering is software design. It comprises discussing requirements, exploring the problem domain, and making design decisions. When globalized, software design could become less effective. Several design activities could be affected, including:

- design modeling (representation),
- design reasoning (about problem domain and solution-domain design aspects), and
- design communication.

Also, lack of awareness (understanding others' activities) and problems with communication media might threaten the success of distributed software design.

Many researchers have explored the impact of distance on collaborative work. James Herbsleb argued that colocation fosters communication because developers are aware of who is around and who is doing what [24]. In contrast, being unable to share resources and see what is happening at the other sites hinders communication across different locations.

Pernille Bjørn and her colleagues investigated whether distance still matters for distributed collaboration [155]. They found that the social challenges form an obstacle to achieving effective work between remote collaborators.

Demetrios Karis and his colleagues performed studies of remote collaboration at Google [65]. They found that the use of videoconferencing and video portals contributes to the success of remote collaboration by:

- providing presence and status information,

- helping to establish mutual trust and common ground, and
- preventing misunderstandings.

However, when it comes to remote design collaboration, Karis and his colleagues highlighted that developers at Google found collaboration over videoconferencing and video portals a pale imitation of face-to-face interaction. Moreover, the developers complained that the video portals at Google lacked a shared drawing tool to facilitate sketching, designing, and brainstorming.

This conforms with what David Budgen stated in his paper "The Cobblers Children": many modeling tools do not serve the purpose of software design and rarely support realistic software design practices [10]. According to Budgen, modeling tools should preserve the flexibility and simplicity of whiteboards and provide proper support for distributed designers at different locations.

Several researchers have proposed next-generation design-support tools that are in line with Budgen's guidelines. One of these tools is OctoUML [83]. OctoUML allows mixed informal modeling (sketching) and formal modeling and supports collaborative distributed software design.

To answer the question posed in our article's title, a deep investigation of current practices of collaborative software design is required. To do so, we analyzed a collaborative design multiple-case study based on two exploratory cases. Details regarding this study are in the following section.

6.2 Multiple-Case Study

In the first case study, three colocated pairs of software designers worked on a software design challenge at a single location (labeled C1, C2, and C3 in the main article) [59]. We conducted the second case study, which involved three design sessions (D1, D2, and D3) between distributed pairs of software design practitioners in Aachen, Germany, and Gothenburg, Sweden. We used the same design problem and timing as in the first case study. This allowed us to:

- explore the design decisions and process activities of the two studies,
- gather experiences and seek insights, and
- develop suggestions and recommendations that could be of interest to practitioners concerned with distributed collaborative software design.

The designers in our study varied from three to seven years of professional experience. Three designers worked in automotive software development, two worked in networking solutions, and one worked in traffic technologies. In both studies, the designers solved a software design challenge. The challenge was to create a software design of a simulator that should enable its users to investigate the effects of different signal timing on traffic flow. The challenge

description is available in *Software Designers in Action: A Human-Centric Look at Design Work* [59].

Teams of two professional software engineers solved the same challenge locally, which focused on four functional requirements:

- Users can create a visual map of intersected roads of varying length.
- Users can describe the behavior of the traffic lights at each of the intersections, such that combinations of individual signals that would result in crashes are prohibited.
- Users can simulate traffic flows on the map, and the resulting traffic levels are conveyed visually.
- Users can change the traffic density per road.

We informed the designers that:

- their design would be evaluated primarily on the basis of its elegance and clarity, and
- they should focus on the interaction that the users will have with the system, including the basic appearance of the program, and on the important design decisions that form the foundation of the implementation.

To create the design, the designers in our case study used a smart whiteboard with the OctoUML[1] collaborative-design software [117] connected to a computer providing videoconferencing between the two sites. OctoUML is open source; it supports mixed informal modeling (free strokes) and formal modeling (UML class diagram shapes) and supports translating free strokes into class diagram shapes on the fly. It provides predefined shapes, drawing selection mechanisms, and undo and redo functionality. With OctoUML, remote designers share a joint canvas upon which they can draw UML diagrams along with informal elements (i.e., text, drawings, etc.). We chose to deploy a simplified version of OctoUML (a shared canvas and sketching tool) on a smart whiteboard that mimics standard whiteboards (see Figure 6.1). Because the designers in the first case study could not use formal modeling, we deactivated those features as well.

Each design session finished with a questionnaire on the experiences and challenges of collaborative distributed design. We analyzed approximately 10 hours of design activity by six pairs of professional software designers and performed a manual coding of more than 2,000 discussion events. For coding the conversation dialogs of the design sessions, we used the collaborative conversation skill taxonomy of Margaret McManus and Robert Aiken [156] and the design-reasoning decisions of Rainer Weinreich and his colleagues [157], as

[1] `http://rodijolak.com/#octouml`

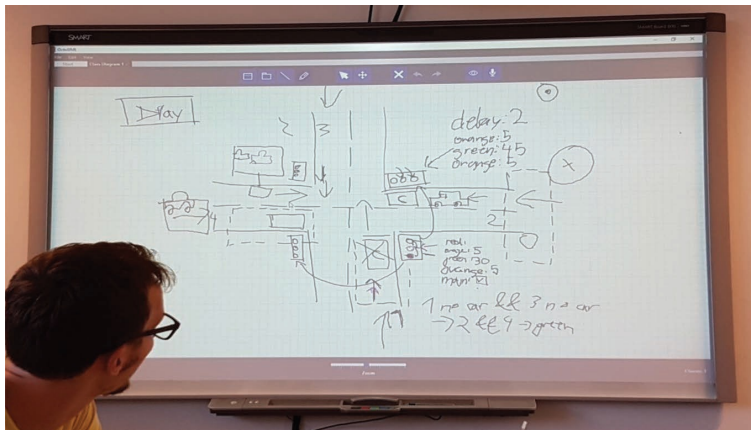


Figure 6.1: UI sketches produced by one of the teams.

presented in Figure 6.2. The former captures various collaborative problem-solving conversation discussions; the latter captures decisions from the problem domain (traffic flow) and solution domain (software engineering). We focused on exploring design reasoning, design communication, awareness, and the number and nature of problems that occurred during the distributed software design sessions.

6.3 How Distance Affects Design Decisions

First, let's look at the type of design decisions that were made and see whether they differed between distributed and colocated design sessions. The graphs in Figure 6.3 indicate that the colocated designers discussed more design decisions in the problem domain than the distributed designers did. More details on how these design decisions were made are available at `http://rodijolak.com/DoesDistanceStillMatter.html`.

The decisions in the problem domain consisted mainly of assumptions, as shown in Figure 6.4. One of the reasons that might have allowed colocated designers to discuss more problem domain design decisions is that they implicitly knew (via facial expressions and body language) whether a specific assumption was mutually understood. In a collaborative process, the conversation can continue only when the collaborators mutually establish what they know [158]. Distance obstructs the process of establishing a mutual understanding of the problem domain between distributed designers. When one designer makes an assumption and implicitly perceives that the colocated partner did not understand that assumption, that designer might rephrase the assumption or build more knowledge around it.

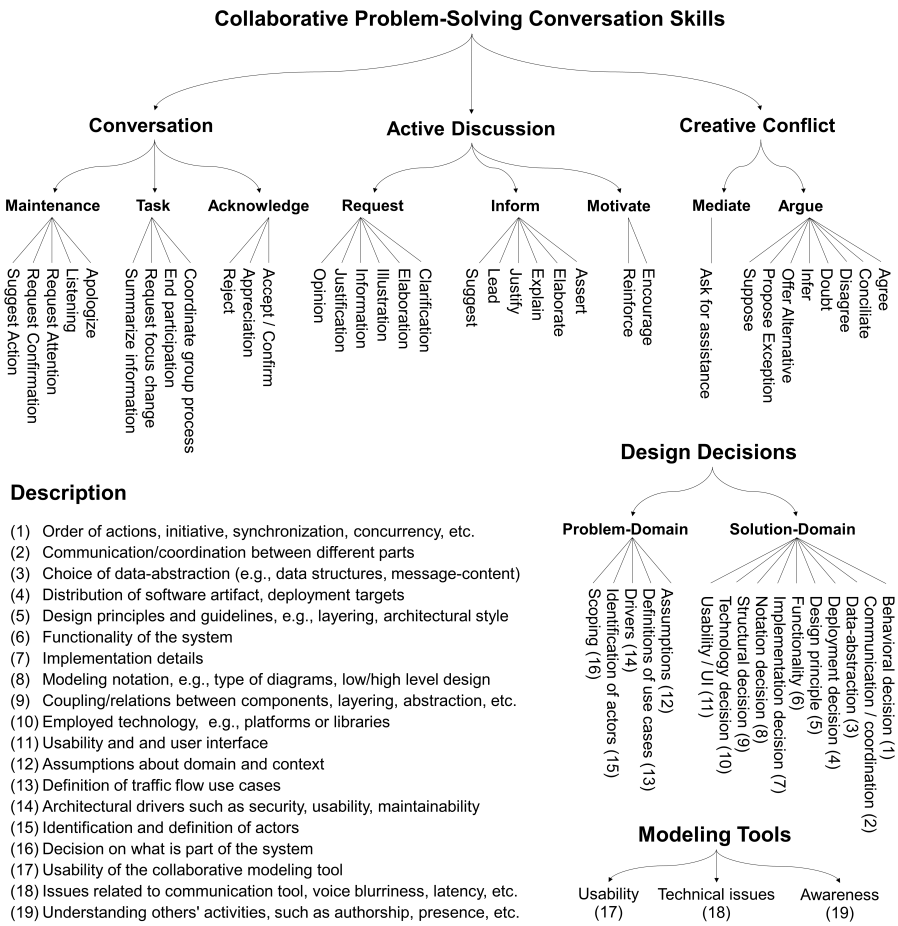


Figure 6.2: A classification schemata for conversation skills and software design decisions [156, 157].

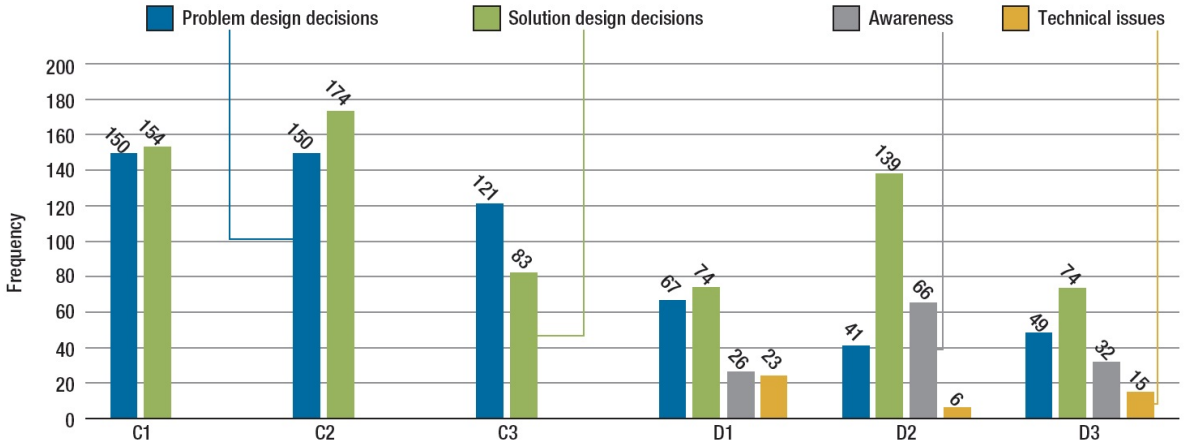


Figure 6.3: The number of design decisions and social and technical issues per each collocated team (C1–C3) and distributed team (D1–D3).

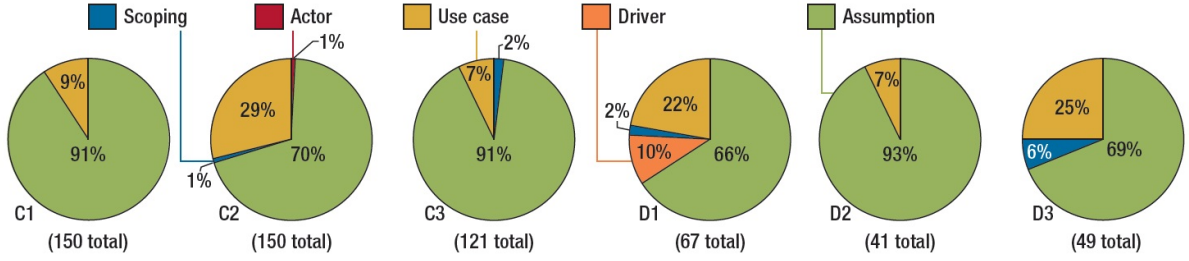


Figure 6.4: The categories of problem domain design decisions made in each design session.

In contrast, distributed designers usually do not see each other when discussing assumptions. Hence, the perception of having a mutual understanding (via body language) was rarely possible. Indeed, the distributed designer making an assumption often implicitly considered that the remote partner understood it, thus producing fewer problem domain design decisions.

Technical issues also affected the distributed design discussions– e.g., through blurriness of the voice and instability of the communication medium. Lack of awareness could have also led to fewer problem domain design decisions in the distributed setting. This is because not perceiving another person's actions makes it difficult to initiate contact and often leads to misunderstanding of communication content and motivation [24].

6.4 How Distance Affects Collaborative Communication

The graphs in Figure 6.5 show how distance could affect communication in distributed design. We see that distributed teams had fewer creative-conflict discussions but more conversation. Creative-conflict discussions can promote software design reasoning and enhance the effectiveness of group tasks [96].

The creative-conflict problem-solving discussion skill comprises two major subskills: Mediate and Argue (see Figure 6.2). Argue comprises different actions (agree, disagree, offer an alternative, propose an exception, etc.). Distributed designers argued less, as shown in Figure 6.6. One of the reasons for fewer arguments–and hence fewer creative-conflict discussions–might have been lack of trust. Colocated designers share experiences and context, which helps them to develop trust. Trust is needed for collaborators to be able to challenge each other without frustrating collaboration. Distributed settings can complicate establishing trust and might compromise reliability between the remote collaborators [159].

Another reason could have been lack of common ground–i.e., the knowledge that the designers are aware of and have in common–in distributed design sessions. When common ground is missing, it might affect distributed collaborators' activities and communication effectiveness [160]. Indeed, this might promote mutual tacit acceptance of design decisions. Hence, it reduces creative-conflict discussions.

Lack of awareness can also reduce creative-conflict discussions. For example, information on authorship (who did what) and intention (what designers are going to do) was not available for the participants in our study.

As we mentioned before, more conversation happened between the distributed designers than between the colocated designers. To explain this, we recall that conversation comprises three major subskills: Maintenance, Task, and Acknowledge (see Figure 6.2). Distant collaboration requires more man-

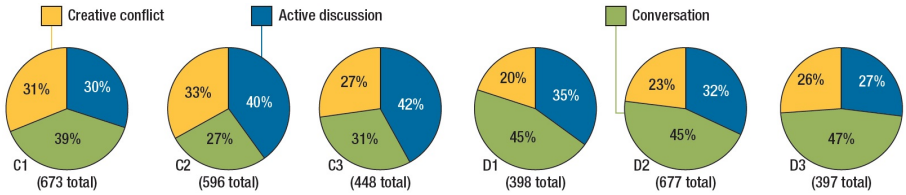


Figure 6.5: The categories of collaborative discussions made in each design session.

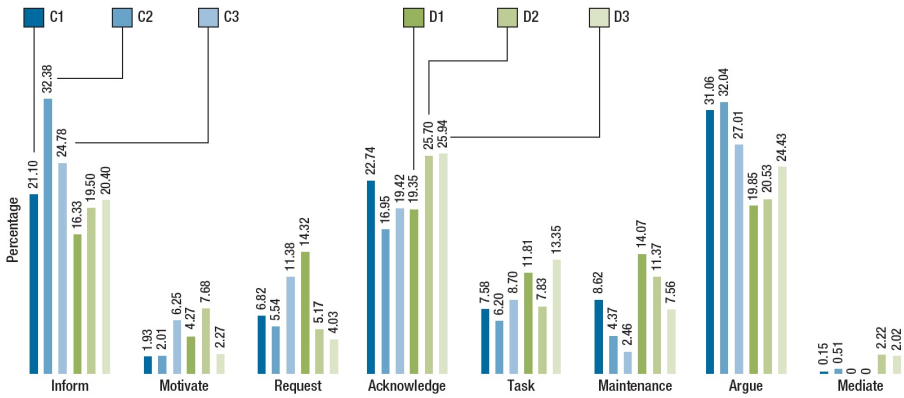


Figure 6.6: Percentages of collaborative-discussion categories per each team.

agement overhead and discussion about work coordination. Lack of awareness among distributed collaborators also raises more task discussions and maintenance discussions. For example, the distributed designers summarized design decisions to confirm knowledge of what they had done so far. Summarizing also helped them understand the intention of their partners.

In addition, distributed designers had fewer Inform (see Figure 6.6) discussions than the colocated designers. This indicates that distributed designers tend to give less information about their decisions, which leads to less active discussion of the essence of and rationale for those decisions.

6.5 The Challenges of Distributed Design

The distributed designers reported the following challenges.

6.5.1 Technological Challenges

The designers considered connection instability as a challenge. Network problems and high CPU use in the client machines interrupted several design sessions. These devices were simultaneously running OctoUML, screen- and voice-recording software, and telecommunication software, which overloaded them. Consequently, the designers had to wait until communication was reestablished. This situation can be prevented by avoiding such overloads.

Moreover, the distributed designers complained about the quality of voice communication. This depends on different factors: the quality of the Internet connection, the distance from the microphone, and the volume of the speakers. This problem can be alleviated by adopting advanced communication infrastructures, a high-speed Internet connection, and advanced voice management tools.

Nonetheless, many organizations fail to keep pace with technological advances and therefore fail to manage the aforementioned challenges.

6.5.2 Social Challenges

First, the designers perceived the lack of awareness as a challenge. In particular, they felt that the inability to interpret eye contact, body language, and facial expressions affected their decisions and activities. For instance, one designer said that because he could not see how his partner reacted to his proposals, he was unable to decide how to act appropriately. Each designer was unaware of what the collaborating designer was doing and which part of the system that designer was talking about or pointing to.

Second, the designers also perceived the lack of trust as a challenge. In particular, the designers felt that not knowing their collaborator beforehand could have affected their discussions and work.

6.5.3 Other Challenges

The design assignment per se was perceived as a challenge. We believe this confirms our process of thoughtfully planning the assignment to simulate real-world software design situations. This planning took into account ideation, problem domain exploration, and design solution decisions.

6.6 Conclusion

The geographical distribution of collaborating partners in practice still raises social and technological challenges. Thus, practitioners should carefully consider whether the distribution is applicable and weigh the benefits of technology deliberately. To support distributed designing, for instance, modern collaborative-design environments focus on the consistent, real-time sharing of

diagrams. However, social awareness, such as the ability of designers to relate to each other through pointing, gaze, and gestures, remains missing.

The designers in our study indicated challenges in distributed collaborations that are beyond the scope of tooling. In particular, in contrast with locally collaborating teams, distributed designers did not know their collaborators beforehand. Hence, they had to build professional and personal trust during the experiment.

On the basis of our results, we encourage software design practitioners aiming to collaborate remotely to consider the following:

- Establish trust via arranging personal or virtual meetings or social events before the remote design sessions [65].
- Establish common ground via exchanging interests, experiences, expertise, and beliefs between distributed designers.
- Introduce explicit triggers for creative-conflict discussions into the collaboration process.

Furthermore, we recommend that the developers of computer-supported cooperative work (CSCW) tools for software design should support awareness by adapting technology to provide immersive telepresence experiences.

Software design requires extensive exploration of the problem domain and context, and leads up to making critical design decisions about software systems. Moreover, collaborative software design is tightly coupled work that requires either more frequent or more complex interactions [154]. Because of these aspects of software design and because the current technology is still incapable of fully mitigating the social challenges of remote collaboration, we suggest that distance still matters.

Chapter 7

Software Engineering Whispers: The Effect of Textual Vs. Graphical Software Design Descriptions on Software Design Communication

(F) R. Jolak, M. Savary-Lelanc, M. Dalibor, A. Wortmann, R. Hebig, J. Vincur, I. Polasek, X. Le Pallec, S. Gérard, M.R.V. Chaudron "Software Engineering Whispers: The Effect of Textual Vs. Graphical Software Design Descriptions on Software Design Communication"
In submission to EMSE Journal. Revised version submitted on Dec. 2019.

Abstract

Context: Software engineering is a social and collaborative activity. Communicating and sharing knowledge between software developers requires much effort. Hence, the quality of communication plays an important role in influencing project success. To better understand the effect of communication on project success, more in-depth empirical studies investigating this phenomenon are needed. *Objective*: We investigate the effect of using a graphical versus textual design description on co-located software design communication. *Method*: Therefore, we conducted a family of experiments involving a mix of 240 software engineering students from four universities. We examined how different design representations (i.e., graphical vs. textual) affect the ability to *Explain*, *Understand*, *Recall*, and *Actively Communicate* knowledge. *Results*: We found that the graphical design description is better than the textual in promoting *Active Discussion* between developers and improving the *Recall* of design details. Furthermore, we found that *well-organized* and *motivated* textual design descriptions help to enhance the explaining and recall of their details.

Keywords: Software Engineering; Software Design; Software Modeling; UML; Communication; Knowledge Sharing; Graphical Representation; Textual Representation; Family of Experiments

7.1 Introduction

Software engineering is a social activity and requires intensive communication and collaboration between developers. In large companies, developers work in different development teams and collaboratively communicate with many stakeholders. In such a setting, the quality of communication between the stakeholders plays an important role in reducing the overall teams', and thus projects', development effort. In a multiple-case study on challenges and efforts of model-based software engineering approaches, Jolak et al. [27] analyzed the distribution of efforts over different development activities in two software engineering projects. Interestingly, they found that communicating and sharing knowledge dominates the effort spent by developers. The effort on communication, as Jolak et al. found, is actually more than all of the efforts that developers spent in any of the other observed development activities, such as, requirements analysis, design, coding, testing, integration, and deployment.

Furthermore, poorly defined software applications (due to miscommunication between stakeholders) can affect the final structure and/or behavior of these applications. This is in line with Jarboe et al. [92] and Kortum et al. [93] who consider that the quality of communication does influence developers' activity experience and achievement, and therefore customer's satisfaction.

The aforementioned studies underline the importance of communication in Software Engineering (SE). They also highlight the need to study communication in-depth to determine elements or criteria of its efficiency and effectiveness. The study we present in this article is inline with this concern: we investigate how different software architecture design representations affect the communication of design knowledge. In particular, we compare textual vs. graphical representations. In contrast with a *textual* representation, a *graphical* representation provides a two-dimensional visuospatial description of information reflecting the actual spatial configurations of the parts of a process or system [56]. With respect to knowledge communication, we look into the following *communication* aspects:

- *Explaining*: or knowledge donating, communicating the personal intellectual capital from one person to others [94].
- *Understanding*: or knowledge collecting, receiving others' intellectual capital [94].
- *Recall*: or memory recall, recognizing or recalling knowledge from memory to produce or retrieve previously learned information [95].
- *Collaborative Interpersonal Communication* [96], which includes:
  - *Active Discussion*: questioning, informing, and motivating others.
  - *Creative Conflict*: arguing and reasoning about others' discussions.

– *Conversation Management*: coordinating and acknowledging communicated information.

7.1.1 Rationale

Kauffeld et al. [161] suggested that effective team communication and information flow are prerequisites for the success of software development projects. In a study on requirements practices in start-ups, Gralha et al. [162] identified knowledge management and communication as increasingly important strategies for risk mitigation and prevention. As a consequence, research concerning different factors influencing the degree and way in which people communicate and share their knowledge is actually relevant for maximizing the aforementioned advantages.

Graphical descriptions encode and present knowledge differently from textual descriptions. In particular, they provide a visuospatial representation of information, and can recraft these information into a multitude of forms by using fundamental graphical elements, such as dots and lines, nodes and links [56]. Moreover, graphical descriptions encourage spatial inferences (e.g., inferences about the behavior, causality, and function of a system) to substitute for and support abstract inferences [163]. This is inline with Moody [28], who states that graphical and textual knowledge representations are differently processed by the human mind. Empirical evidence on how graphical descriptions affect developer's achievement and development productivity is still underwhelming, as reported by Hutchinson et al. [5]. Moreover, Meliá et al. [29] report that the software engineering field lacks a body of empirical knowledge on how different representations (graphical vs. textual) could provide support for improving software quality and development productivity.

In this study, we focus on design knowledge communication/transfer between two software developers, where, by using a graphical vs. textual software design description, one developer is taking the role of design *Explainer* (i.e., design knowledge owner), and one developer is taking the role of design *Receiver* (i.e., design knowledge receiver). Rus et al. [164] reported that greatest challenge of companies is to retain tacit knowledge, mainly, but also explicit knowledge (e.g., models).

Companies, such as *Ericsson Software Technology*[1] and *sd&m*[2], started initiatives –Ericsson's initiative is called "*Experience Engine*"– to exchange knowledge between developers by connecting two individuals, a *problem owner* and *experience communicator*. The problem owner is the employee who requires information or support to solve a specific problem and the experience communicator is the employee who has in-depth knowledge of the problem domain. Having been connected, the experience communicator has to *educate*

[1] `https://www.ericsson.com`
[2] `https://www.capgemini.com`

the problem's owner on how to solve it. The aforementioned initiatives illustrate that our study has a practical relevance.

7.1.2 Objective and Contribution

We planned and conducted a family of experiments with a goal to *understand* and *compare* the effect of using a Graphical Software Design description (GSD) *versus* a Textual Software Design description (TSD) on software design communication. Through this, we contribute to the body of empirical knowledge on the practical use of graphical versus textual software design descriptions. Such knowledge might lead to achieving more effective software design communication, which in turn would help in reducing the total effort of software development activities. Consequently, we address the research objective by answering the following question:

- **R.Q.1** How does the representation of software design (graphical vs. textual) influence *[Communication Aspect]*?

  Where the investigated *[Communication Aspect]s* are the following:

  - Design Explaining,
  - Design Understanding,
  - Design Recall,
  - Active Discussion,
  - Creative Conflict, and
  - Conversation Management.

We first *understand* how each software design representation (i.e., graphical/textual) affect the six aspects of communication that we described previously (i.e., explaining, understanding, recall, active discussion, creative conflicts, and conversation management). Then, we *compare* the effect of using the graphical vs. textual software design description on the considered communication aspects.

To address certain threats to external validity, we also compare the effect of using a *cohesive*[3] and *motivated*[4] TSD versus less cohesive and unmotivated TSD on software design communication. In particular, we address the following research question:

- **R.Q.2** Does using a cohesive and motivated TSD influence *[Communication Aspect]*?

  Where the investigated *[Communication Aspect]s* are the following:

[3] *Cohesive*: documented information or knowledge that are well-organized.
[4] *Motivated*: augmented with design rationale.

- Design Explaining,
- Design Understanding,
- Design Recall,
- Active Discussion,
- Creative Conflict, and
- Conversation Management.

The remainder of this paper is organized as follows: We discuss the related work in Section 7.2. We describe the family of experiments in Section 7.3. We present the results in Section 7.4. We discuss the results and threats to validity in Section 7.5. Finally, we conclude and describe the future work in Section 7.6.

7.2 Related Work

Effective communication depends on various factors, such as personality [165], distance [97], or knowledge representation (graphical vs. textual) [29, 66, 166].

In a recent study on design activities of co-located and distributed collaborative software design [97], the authors investigate whether advanced technologies for distributed communication can replace personal meetings. The main result is that co-located face-to-face meetings remain relevant as facial reactions and body language are often not transmitted by current communication software. This is partly due to technical challenges, such as unstable or slow Internet connection, that affect communication results. In contrast to that study, our family of experiments does not investigate distributed communication and is conducted without the use of communication tool-support to mitigate the effects of technical challenges.

Meliá et al. [29] describe an experiment in which students perform maintenance tasks on a graphical model and on a textual model. The authors investigate whether a model's syntax affects subjective and objective performance and whether the notation influences developer satisfaction. Objective performance is measured by the number of correct answers in the task whereas the subjective performance is the performance as perceived by the developer. In the experiment, participants were divided into two groups, one group worked with a model for selling tickets, the other group had a model for organizing online courses. Participants received the models in textual and in a graphical notation and were asked to find 5 errors in each notation. They also received 5 tasks in which they had to extend and modify each model. Participants using the textual notation performed significantly better in finding errors in the domain model and also spent less time until finishing the task. Nonetheless, participants preferred to work with the graphical notation. The authors believe this to originate from the fact that students learn graphical modelling languages

such as UML as stereotypes for domain models whereas less attention is given to textual modelling languages.

In another study, the authors measure how well participants extract the required information (such as architectural design decisions) from different media [66]. The researchers collected participant-specific information in two questionnaires, filmed participants during tasks, and asked them to think out loud. The experiment comprised of four architectures, out of which each consisted of a graphical and a textual description. Participants (students and professional developers) were asked three questions per architecture. The authors observed that no notation was clearly superior in communicating architecture design decisions. Nonetheless, participants tended to first look at the graphical notation before reading the text. The authors attribute this to the clarity of the graphic representation, which enables participants to grasp the structure of the model more quickly.

In a case study on comparing graphical versus textual representations, the researchers measure the accuracy and time spent to solve three requirement comprehension tasks [166]. The study does not indicate results concerning accuracy (both notations yield correct results), but participants spent less time when working with textual requirements. Participants preferred to work with the graphical representation nonetheless. Also, when working with a combination of graphical and textual representations, the study measured the best results concerning time and accuracy.

Other research investigated a combined usage of textual and graphical representations [167]. The researcher interviewed 21 practitioners to find out how developers work with different requirements artifacts of various granularity and notation and how they handle scattered information. The researcher found out which artifacts the practitioners used and which problems they encountered. A share of 70% of the interviewees reported issues when working with multiple artifacts. The main shortcomings were inconsistencies and the additional effort for documenting.

In contrast to our research, the studies described in [29, 66, 166, 167] do not observe communication based on using graphical and textual representations, but how well both notations are suited to share information. Therefore, more research concerning both notations as a base for explaining and discussing software architectures is required.

Other related research aims to find out if drawing improves the recall ability, compared to making textual notes [168]. The participants of this research were divided into two groups, younger and older adults, to measure whether the notation influences both groups in the same way. Participants were told 30 nouns, one after the other, and asked to either draw or to note the items textually. Afterwards, they were asked to recall and list all items. For the drawn nouns both groups performed equally well, but for the textual words, young adults performed better. This indicates that a graphical notation can

compensate for the age-related deficit.

To summarize the related work section, Table 7.1 provides a brief comparison between the research objectives of our study and related work.

Table 7.1: Research objectives of our study and related work.

Work	Objectives
Heijstek et al. [66]	Study the effect of using a graphical vs. textual on extracting design decisions information.
Jolak et al. [97]	Study the effect of distance on communication
Liskin [167]	Study the use of different requirement artifacts in practice.
Meade et al. [168]	Study the effect of drawings vs. textual notes on memory recall.
Meliá et al. [29]	Study the effect of using a graphical vs. textual notation on domain models maintenance.
Sharafi et al. [166]	Study the effect of using a graphical vs. textual representation on requirement comprehension.
Our Study	*Study the effect of using a graphical vs. textual software design description on face-to-face design communication.*

7.3 Experimental Design

This section describes the protocol that is used to perform the experiments and analyze the results. In particular, we report the experiment according to the guidelines suggested by Jedlitschka et al. [169].

7.3.1 Family of Experiments

Easterbrook et al. [77] highlighted that controlled experiments help to investigate testable hypotheses where one or more independent variables are manipulated to measure their effect on one or more dependent variables. A family of experiments facilitates building knowledge and extracting significant conclusions through the execution of multiple experiments pursuing the same goal. Basili et al. [170] reported that families of experiments can help to (i) generalize findings across studies and (ii) contribute to important and relevant hypotheses that may not be suggested by individual experiments.

We planned and conducted a family of experiments based on the methodology of Wohlin et al. [76]. Our family of experiments are between-subject designs to minimize learning effects and transfer across conditions. The family of experiments consists of one original experiment and three external replications involving 240 participants in total (See Table 7.2). The original experiment (OExp) was conducted at the University of Gothenburg involving a mix of 50 B.Sc. and M.Sc. Software Engineering students. The first replication (REP1) was conducted at RWTH Aachen University with 36 M.Sc. and Ph.D. SE students. The second replication (REP2) was conducted at the University of

Lille involving 94 M.Sc. SE students. REP1 and REP2 replicated as accurately as possible the original experiment (*strict replications* [170]). The third replication (REP3) was conducted at the Slovak University of Technology with 60 B.Sc. and M.Sc. SE students. REP3 varied the manner in which the original experiment was conducted, so that certain threats to external validity were addressed. In particular, REP3 is a replication that varies a variable intrinsic to the object of study [170]. This variable is the TSD. More details regarding this variation are provided in Section 7.3.5.

The experiment material and communication language in OExp, REP1, and REP3 were in *English*. In contrast, the experimental material and communication language in REP2 (which was conducted at the University of Lille, France) were in *French*. The gender distribution in each experiment is also shown in Table 7.2. The majority of the participants are males (79%).

Table 7.2: The family of experiments.

ExpID (R.Q.)	S.R.	Context	Lang.	Date	Participants	#	Females
OExp (R.Q.1)	-	Chalmers & Gothenburg University	English	11/10/18	B.Sc. & M.Sc. Students	50	22%
REP1 (R.Q.1)	Yes	RWTH Aachen University	English	25/10/18	M.Sc. & Ph.D. Students	36	17%
REP2 (R.Q.1)	Yes	University of Lille	French	03/12/18	M.Sc. Students	94	26%
REP3 (R.Q.2)	No	Slovak University of Technology	English	13/12/18	M.Sc. Students	60	15%
					Total	240	21%

R.Q.: Research Question
S.R.: Strict Replication

7.3.2 Scope

Developers intensively communicate ideas, decisions, progress, updates, etc. throughout the software development life-cycle. In this study, we focus on investigating co-located, face-to-face software design communication.

Design communication plays a fundamental role in transferring software design decisions (i.e., instructions for software construction) from *architects/analysts* to *programmers* or other stakeholders. Also, the quality of these communications might play an important role in shaping the overall structure and behaviour of software products.

In co-located teams, developers usually communicate face-to-face. In distributed teams, developers use other communication channels, such as video conferencing systems. Jolak et al. [97] found that co-located software design discussions are more effective than distributed design discussions. Moreover,

Storey et al. [171] stated that face-to-face communication is one of the most important and preferred communication channel for collaborative software development. Indeed, with face-to-face communications, developers can receive feedback quickly which facilitates discussing through complex issues, such as design decisions. Thus, we investigate co-located face-to-face communication, as this type of communication is widely preferred and would therefore contribute to the generalizability of our results.

Modeling languages can be (i) of general-purpose and applied to any domain, such as the Unified Modeling Language (UML) or (ii) domain-specific and designed for a specific domain or context, such as the Domain Specific Languages (DSLs). Brambilla et al. [123] stated that UML is widely known and adapted, and comprises a set of different diagrams for describing a system from different perspectives. Brian Dobing and Jeffrey Parsons [172] found that the use of UML class diagrams substantially exceeds the use of any other UML diagram (use case, sequence, activity, etc.). Thus, in order to increase the generalizability of the results of this study we chose to represent the graphical software design description by a UML class diagram.

7.3.3 Participants

The population for this study was intended to match two prerequisites: (i) having a basic knowledge in UML (especially UML class models), (ii) and being able to understand and communicate in the experiment language. The target group in this case is the entire group of people who posses the aforementioned criteria: students who took an academic course in UML modeling, professional software developers, architects, etc. However, the portion of the population to which we had reasonable access is a subset of the target population. In particular, the accessible population for this family of experiments was the group of B.Sc. and M.Sc. Software Engineering (SE) students at the universities where the authors teach SE courses. The sampling approach was *convenience sampling*. On the one hand, this sampling approach is easy and readily available. On the other hand, the sample produced by convenience sampling might not represent the entire population (i.e., threat to external validity or generalizability of the results). To increase the external validity of the results, we recruited a mix of *240* B.Sc. and M.Sc. SE students from four universities to take a part in a family of experiments. Previously in Section 7.3.1 (Table 7.2), we provided details on the participants in this family of experiments.

7.3.4 Experimental Treatments

The participants of each experiment were *randomly* assigned to two treatments or groups:

- **Group G**: participants in this group had to discuss a software design as

represented by a graphical description (UML class diagram).

- **Group T**: participants in this group had to discuss the same software design, but as represented by a textual description.

Furthermore, the participants of each group were *randomly* assigned one specific role:

- **Explainer**: this role consisted in: (i) understanding the design representation, and (ii) explaining it to a *Receiver*.
- **Receiver**: this role consisted in understanding the software design based on the discussion with an *Explainer*.

Having the roles assigned, we *randomly* formed 120 *Explainer-Receiver* pairs. These pairs were involved in discussing a design case which we detail in the next Section 7.3.5.

7.3.5 Design Case and Graphical vs. Textual Descriptions

We created a design case for our family of experiments. The design case describes a structural view of a mobile application of a fitness center, the *Fitness Paradise*. This fitness center gives its clients the opportunity to book facilities and activities. The featured application enables clients to consult the schedule of activities, manage bookings, keep track of payments, and visualize performance data when available. We believe that the selected design case relies on a familiar domain, *Sport and Gym*, from everyday life which is quite popular and easy to understand without prior knowledge.

To introduce the *Explainers* with the design case, we created a *design case specification* document which describes the fitness center and lists the features of the mobile application in natural language:

Design Case Specification
The Fitness Paradise is offering a combination of sport activities. To this end, it has several facilities: an Olympic size swimming pool, a gym with several fitness machines (weight, rowing, cycles), a football-pitch, a tennis court. The Fitness Paradise offers these facilities for rent for its clients. The Fitness Paradise accepts both individuals and clubs (e.g., companies and sport clubs) as members. The members have an account. In addition to the facilities, it also offers activities: training, yoga classes, physiotherapy and massages. The activities have a schedule that is updated every month. The Fitness Paradise wants to support booking via an app for smart phones. The app presents the activity schedule for every day and week. Bookings can be paid with different methods. At the end of every month, an invoice is sent to members for any unpaid

bookings over the past month. The system keeps track of the bookings. The app can also keep track of the fitness and performance of Fitness Paradise members. Examples of performance are for example: the distance cycled in a certain amount of time, calories burnt in a training session (but also aggregates over a week and month). For the fitness enthusiasts it is also possible to exchange such performance data which is collected by the member's fitness-band/watch.

7.3.5.1 Design Descriptions

We played the role of the *designer/architect* and created the two design representations (i.e., GSD and TSD). The GSD and TSD provide the same information and describe one structural design of the *Fitness Paradise* app. The two design descriptions only differ in the way they represent the design (i.e., graphical vs. textual).

- **GSD**. We created the UML class diagram of the design case (see Figure 7.1). The diagram includes 28 classes (21 model entities, 3 controllers, and 4 views) and 30 relationships. We chose to use the Model View Controller (MVC) design pattern for structuring the design, as this pattern is well known by the participants of the experiments. The entities of each part of the MVC were given a specific color. The model entities have a yellow color, the controllers are in blue, and the views are in green. The colors were added to the entities in the GSD in order to mimic the characteristic of structured textual document (which we describe next) in facilitating a visual distinction between different sections (i.e., the three parts of the MVC).

- **TSD**. For EXP, REP1, and REP2, each element of the GSD was methodologically used to create exactly one corresponding element (e.g., one paragraph or sentence) in the textual description, thus to maintain a one-to-one correspondence between GSD and TSD (See Figure 7.2). In other words, we were thoroughly keen to make both the graphical and textual designs present the exact same amount of information or design knowledge in order to control eventual bias due to different amount of information. The textual description was arranged into two main structured sections. In the first section, we orderly described the entities of each module of the MVC: First the entities of the model part, then the entities of the controller part, and last the entities of the view part. In the second section, we described the relationships between the entities following the same appearance order of the entities.

- **Altered TSD**. In REP3, we wanted to know whether or not a *motivated* and *cohesive* TSD could affect design communication differently from the original TSD. Figure 7.3 shows the altered TSD.

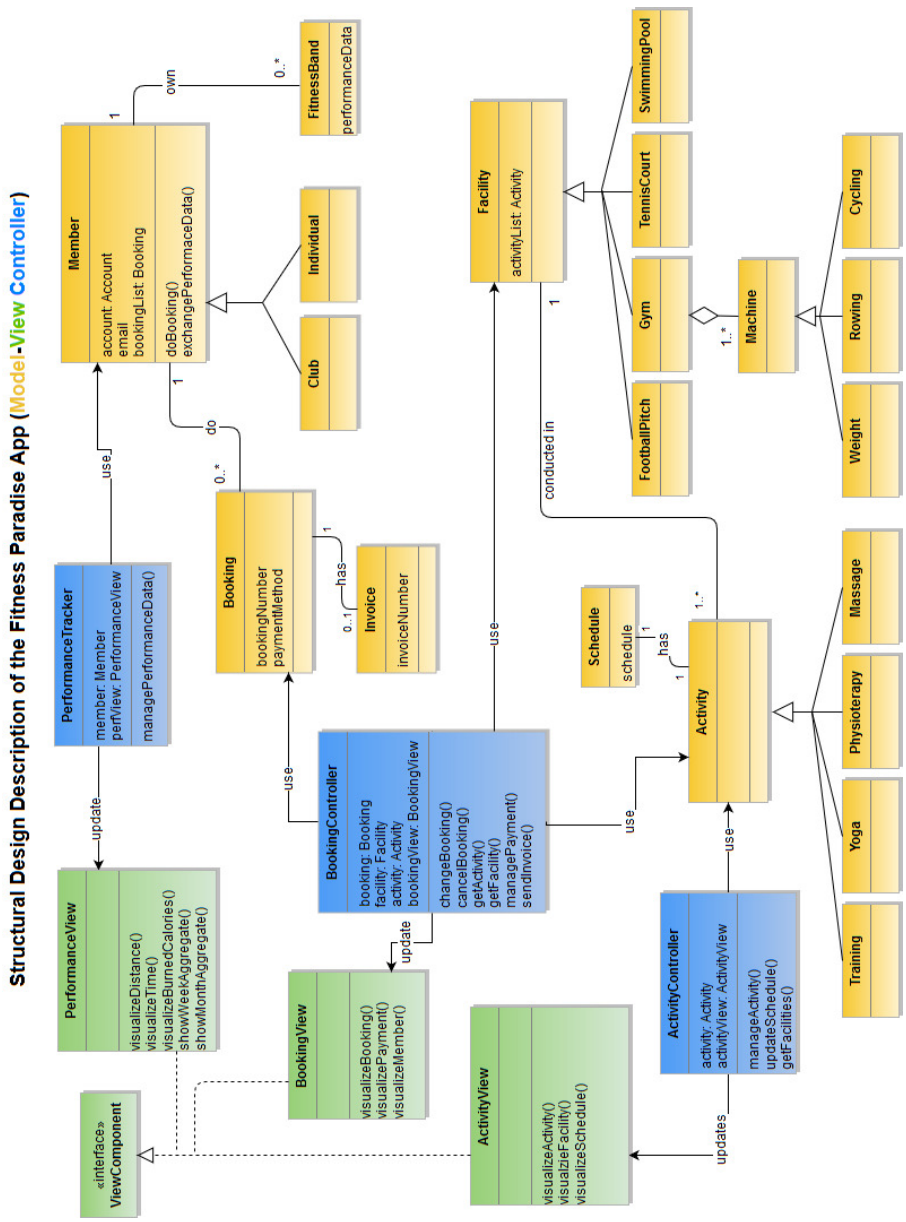


Figure 7.1: Graphical Software Design Description (GSD)

Structural Design Description of the Fitness Paradise App

The software of the app is designed according to the Model-View Controller (MVC) design pattern.

1. ENTITIES

I. Model

The model part includes of the following entities:

- **Member:** this entity provides information on user account, email and the list of bookings done by one member. Via this entity the member can do bookings and exchange performance data. The member can be a club or an individual.
- **FitnessBand:** this entity provides information on the performance data of the member.
- **Booking:** this entity provides information on the booking number and the payment method.
- **Invoice:** this entity provides information on the invoice number.
- **Facility:** this entity provides information on the list of activities conducted in one facility. The facility can be a football pitch, tennis court, swimming pool or gym. In the gym, there is at least one machine. The machine can be a: weight, rowing or cycling machine.
- **Activity:** This entity can be training, yoga classes, physiotherapy sessions, or massage sessions.
- **Schedule:** this entity provides information on the schedule of the activities.

II. Controller

The controller part includes the following controllers:

- **Booking Controller:** this controller uses the data from the Booking, Facility and Activity entities. This controller updates the Booking View. The functionality of this controller is to change the booking, cancel the booking, get a facility, get an activity, manage the payments, and send the invoices.
- **Activity Controller:** this controller uses the data from the Activity entity. This controller updates the Activity View. The functionality of this controller is to manage the activities, update the schedule of the activities, and get the facilities where the activities are taking place.
- **Performance Tracker:** this controller uses the data from the Member entity. This controller updates the Performance View. The functionality of this controller is to manage performance data of the member.

III. View

The view part includes three views that implement the *View Component* interface:

- **Booking View:** this view provides options to visualize the booking, visualize the payment and visualize the member.
- **Activity View:** this view provides option to visualize the activity, visualize the facility and visualize the schedule of the activities.
- **Performance View:** this view provides option to visualize the covered distance and time, the burned calories, but also aggregates over a week and month.

2. RELATIONS

- **Member:**
  - One Member can do zero or many Bookings.
  - One Member owns zero or many FitnessBands.
  - One Member can be:
    - Club
    - Individual
  - The Member entity is used by the Performance Tracker.
- **FitnessBand:**
  - One FitnessBand belongs to one Member.
- **Booking:**
  - One Booking is done by one Member
  - One Booking has zero or one Invoice.
  - The Booking entity is used by the Booking Controller.
- **Invoice:**
  - One Invoice is related to one Booking.
- **Facility:**
  - One Facility can host one or many Activities.
  - One Facility can be:
    - Football pitch
    - Tennis court
    - Swimming Pool
    - Gym, which consists of one to many Machines, and one Machine can be:
      - Weight
      - Rowing
      - Cycling
  - The Facility entity is used by the Booking Controller.
- **Activity:**
  - One Activity has one Schedule.
  - One Activity is conducted in one Facility.
  - One Activity ca be:
    - Training
    - Yoga class
    - Physiotherapy session
    - Massage session
  - The Activity entity is used by the Booking Controller.
- **Schedule:**
  - One Schedule belongs to one Activity.
- **Booking Controller:**
  - This controller uses the Booking, Activity and Facility entities.
  - This controller updates the Booking View.
- **Activity Controller:**
  - This controller uses the Activity entity.
  - This controller updates the Activity View.
- **Performance Tracker:**
  - This controller uses the Member entity.
  - This controller updates the Performance View.
- **Booking, Activity and Performance Views:**
  - These controllers implement the interface "View Component".

1 2 3

Figure 7.2: Original Textual Software Design Description (TSD)

Structural Design Description of the Fitness Paradise App

1. Purpose

This *Software Design Specification* is made to provide a blueprint of the software design of the FPA. This document provides a textual description to help developers to get a deeper insight into the structure of the software.

2. Intended audience

This document is directed towards the development team and the customer (i.e., the fitness paradise center's owner).

3. System Overview

The FPA is a software that facilitates the booking and use of the facilities of the fitness paradise center. Also, the FPA allows its members to visualize information on the booked facilities and activities, and to share performance data with other members.

4. System Design

The software of the app is designed according to the Model-View Controller (MVC) design pattern to have a: (i) faster development process: MVC supports rapid and parallel development, (ii) ability to provide multiple views: in the MVC Model, you can create multiple views for a model. (iii) better support for maintenance: modification does not affect the entire model because model part does not depend on the views part.

The Model is the central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application. The Model part consists of several entities:

- **Member**: can be a club or an individual. A member has information on user account, email and the list of bookings that he/she did. The member can do bookings and exchange performance data. One Member can do zero or many Bookings. One Member owns zero or many FitnessBands. The Member entity is used by the PerformanceTracker.
- **FitnessBand:** provides information on the performance data of the member. One FitnessBand belongs to one Member.
- **Booking:** provides information on the booking number and the payment method. One Booking is done by one Member. One Booking has zero or one Invoice. The Booking entity is used by the Booking Controller.
- **Invoice:** provides information on the invoice number. One Invoice is related to one Booking.
- **Facility:** provides information on the list of activities conducted in one facility. One Facility can host one or many Activities. The facility can be a football pitch, tennis court, swimming pool or gym. In the gym, there is at least one machine. The machine can be a: weight, rowing or cycling machine. The Facility entity is used by the Booking Controller.

1

- **Activity:** can be training, yoga classes, physiotherapy sessions, or massage sessions. One Activity has one Schedule. One Activity is conducted in one Facility. The Activity entity is used by the Booking Controller.
- **Schedule:** provides information on the schedule of the activities. One Schedule belongs to one Activity.

The Controller part consists of three controllers that accepts input and converts it to commands for the model or view:

- **BookingController:** this controller uses the data from the Booking, Facility and Activity entities. This controller updates the Booking View. The functionality of this controller is to change the booking, cancel the booking, get a facility, get an activity, manage the payments, and send the invoices.
- **ActivityController:** this controller uses the data from the Activity entity. This controller updates the Activity View. The functionality of this controller is to manage the activities, update the schedule of the activities, and get the facilities where the activities are taking place.
- **PerformanceTracker:** this controller uses the data from the Member entity. This controller updates the Performance View. The functionality of this controller is to manage performance data of the member.

The view can be any output representation of information. The view part includes three views that each implement a *ViewComponent* interface:

- **BookingView:** this view provides options to visualize the booking, visualize the payment and visualize the member.
- **ActivityView:** this view provides option to visualize the activity, visualize the facility and visualize the schedule of the activities.
- **PerformanceView:** this view provides option to visualize the covered distance and time, the burned calories, but also aggregates over a week and month.

2

Figure 7.3: Altered Textual Software Design Description (Altered TSD)

To make the design *motivated*, we added an introduction to the original textual description, including a rationale of the chosen design pattern (i.e., MVC).

To make the design *cohesive*, we organized the textual design description

in a different way. In particular, the description of the relationships of each entity was moved and placed right after the description of the entity. In this way, information regarding each entity and its relationships are close to each other, instead of being distant/remote (i.e., located on different pages), as in the original textual description.

7.3.6 Tasks

The main task of this family of experiments was inspired by the Chinese Whispers (or The Telephone) game. In this game, players form a line, and the first player comes up with a message and whispers it to the ear of the second person in the line. The second player repeats the message to the third player, and so on. When the last player is reached, they announce the message they heard to the entire group.

In contrast, we created a message (i.e., a software design representation), and asked the first player (i.e., the *Explainer*) to first understand the message then explain it to the second player (i.e., the *Receiver*). After that, the players (i.e., *Explainers* and *Receivers*) have to announce the message (i.e., via answering a post-task questionnaire). Finally, the original message (i.e., the software design representations) is compared with the final version (i.e., knowledge of *Explainers* and *Receivers*).

The main task of the experiments reflects common scenarios in software engineering industry where developers collaborate, communicate, and exchange knowledge in order to create software. For example, the main task reflects a common knowledge-transfer scenario between a software architect (i.e., Expaliner) who owns knowledge on the structure and behavior of the software system and a software developer (i.e., Receiver) who needs to receive and understand the knowledge of the architect in order to start coding. Moreover, knowledge communication is especially important when new employees enter a company and *struggle* to learn the existing tacit knowledge. In this direction, our task reflects the scenario of onboarding of novice developers by experienced developers (e.g., Ericsson's "*Experience Engine*" initiative (cf. section 7.1).

In addition to the main task, we added two secondary tasks to collect complementary data, such as participants' design experience and communication skills that are also needed for data analysis and results' discussion.

For the study, the participants had to perform the following three tasks:

(A) **Answer the pre-task questionnaire**. All participants have to answer the pre-task questionnaire based on the group they are assigned to. No time-limit is imposed for this task. We noted the required time for this step during the experiments and found that it takes 15 minutes on average.

The pre-task questionnaire is developed to collect participants' design experiences and communication skills based on self-evaluations. The

questions in the pre-task questionnaire vary according to the role of the participant (*Explainer* vs. *Receiver*) and his/her group (G vs. T).

- **G-Explainer**: participants belonging to this subset have to answer questions on (i) familiarity with software design and UML modeling, (ii) how good are they in understanding and *sense-making*[5] of UML models, an English/(French) conversation, and explaining their knowledge to others.
- **G-Receiver**: participants belonging to this subset have to answer questions on (i) familiarity with software design and UML modeling, (ii) how good are they in understanding and sense-making of UML models, an English/(French) conversation, and *building* knowledge from conversing with others.
- **T-Explainer**: participants belonging to this subset have to answer questions on (i) familiarity with software design, (ii) how good are they in reading, understanding, and sense-making of an English/French text, understanding and sense-making of an English/French conversation, and explaining their knowledge to others.
- **T-Receiver**: participants belonging to this subset have to answer questions on (i) familiarity with software design, (ii) how good are they in understanding and sense-making of an English/French conversation, and *building* knowledge from conversing with others.

(B) **Discuss the Design (i.e., transfer design knowledge)**. Each *Explainer* receives a *design case specification* plus either a GSD or TSD, based on *Explainer*'s group (G or T). The *Explainer* has to read and understand the received artifacts, as good as he/she can, in 20 minutes (defined based on to the pilot studies, see Section 7.3.7.1). The *Explainers* are allowed to individually ask questions to experiment supervisors to clarify issues related to the design, if required.

After 20 minutes, the *Explainers* give the *design case specification* back to the supervisors, but keep the design description (GSD or TSD). Each *Explainer* is randomly paired with a *Receiver* from the same group. Then, each *Explainer-Receiver* pair is given 12 minutes (defined based on to the pilot studies, see Section 7.3.7.1) to discuss the design, where the *Explainer* has to explain the design and the *Receiver* has to understand the design. The *Receivers* can unhesitatingly ask questions. Moreover to help the understanding process, *Receivers* are allowed to take notes during the discussion, but all notes are collected by the supervisors

[5]Developing the understanding of a concept by connecting it with existing knowledge

before the next task. This is because two of the communication aspects, *Understanding* and *Recall*, that we measure require the participants to, respectively, *apply* and *remember* the design knowledge without using the design descriptions or the notes that they took during the discussions.

(C) **Answer the post-task questionnaire**. All participants have to answer the post-task questionnaire based on their groups. No time-limit is imposed for this task. We also noted the required time for this step and found that it takes 15 minutes on average.

The first part of the post-task questionnaire is developed to collect participants' self-evaluations of their experiences just after the design discussion. The questions vary according to the role of the participant (*Explainer* vs. *Receiver*) and his/her group (G vs. T).

- **G-Explainer**: participants belonging to this subset have to answer questions on (i) how good they are in remembering UML models, (ii) how well they did understand and explain the design, and (iii) how much diagrams did help them in understanding and explaining the design.
- **G-Receiver**: participants belonging to this subset have to answer questions on (i) how good they are in remembering UML diagrams, (ii) how well they did understand the design from the discussion with the *Explainer*, and (iii) how much diagrams did help them in understanding the design.
- **T-Explainer**: participants belonging to this subset have to answer questions on (i) how good they are in remembering a English/French text, (ii) how well they did understand and explain the design, and (iii) in case they could have used them, how much diagrams would have helped in understanding and explaining the design.
- **T-Receiver**: participants belonging to this subset have to answer questions on (i) how good they are in remembering a English/French text, (ii) how well they did understand the design from the discussion with the *Explainer*, and (iii) in case they could have used them, how much diagrams would have helped in understanding the design.

The second part of the post-task questionnaire evaluates participants' understanding and recall abilities.

To measure the *Recall*, we formulated ten questions[6] challenging participants' recall abilities. Two of these questions are open requiring free-text answers, six questions are multiple-choice questions which require the

[6] `http://rodijolak.com/SE_Whispers/Recall_Q.pdf`

participants to choose only one choice, and two questions are check-boxes questions which require to select one or more answers from the available.

To measure the *Understanding*, we formulated three questions[7] focusing on MVC design maintenance (using maintenance questions to measure understanding is motivated in Section 7.4.3). In each question we introduce a design maintenance (i.e., evolution) scenario and suggest four ways to address it. The three questions are multiple-choice questions which require the participants to choose only one choice from 4 provided choices.

To evaluate the answers of the participants on recall and understanding questions, we defined grading rules that can be consulted online[8].

Remark. In REP2 (University of Lille), the pre- and post-task questionnaires, design case specification, GSD, and TSD were translated to *French*, as the SE course that the participants are frequenting is in French. During the translation process, each word was carefully chosen to match the semantics of the original *English* textual description as close as possible. To maintain a strict replication, after the translation process we thoroughly did review the aforementioned artifacts and ensured that the amount of information/knowledge they convey is the same as provided by the artifacts used in the original experiment (OExp).

7.3.7 Variables and Hypotheses

The goal of this study is to compare between the effect of using GSD versus TSD on software design communication. The only independent variable and manipulated factor is the design description. This variable is nominal and corresponds to two treatments/groups: G group using GSD and T group using TSD. In this study, we consider six dependent variables (See Tables 7.3). These variables correspond to the six communication aspects which we described in the introduction.

The original experiment and replications were conducted under the same environment conditions and by following a well-defined protocol to ensure that the impact of any other variable on the results is relatively negligible.

For OExp, REP1, and REP2, we formulate and study the following *null* $\mathbf{H}_0$ and alternative hypotheses $\mathbf{H}_0^a$:

- $\mathbf{H}\text{EXP}_0^a$: The design description $[\textit{has no impact}]_0/[\text{has impact}]_a$ on EXP.
- $\mathbf{H}\text{UND}_0^a$: The design description $[\textit{has no impact}]_0/[\text{has impact}]_a$ on UND.
- $\mathbf{H}\text{REC}_0^a$: The design description $[\textit{has no impact}]_0/[\text{has impact}]_a$ on REC.

[7] `http://rodijolak.com/SE_Whispers/Understanding_Q.pdf`
[8] `http://rodijolak.com/SE_Whispers/Grading_Rules.pdf`

Table 7.3: Dependent variables and measurement.

Dependent Variable	Measure	Source	Measurement Instrument	Measurement Scale
Explaining (EXP)	Ordinal	Subjective	Questionnaire (Perceptions)	5-point Likert Scale Very Poor - Very Good
Understanding (UND)	Interval	Objective	Maintenance Questions	Total Score from 0 to Max 3 Points
Recall (REC)	Interval	Objective	Recall Questions	Total Score from 0 to Max 10 Points
Active Discussion (AD)	Ratio	Objective	Counting Occurrences in Recorded Conversations	AD/(AD+CC+CM) Values from 0 to 1
Creative Conflict (CC)	Ratio	Objective	Counting Occurrences in Recorded Conversations	CC/(AD+CC+CM) Values from 0 to 1
Conversation Mgt. (CM)	Ratio	Objective	Counting Occurrences in Recorded Conversations	CM/(AD+CC+CM) Values from 0 to 1

- $\mathbf{H}\text{AD}_0^a$: The design description $[\textit{has no impact}]_0$/[has impact]$_a$ on AD.
- $\mathbf{H}\text{CC}_0^a$: The design description $[\textit{has no impact}]_0$/[has impact]$_a$ on CC.
- $\mathbf{H}\text{CM}_0^a$: The design description $[\textit{has no impact}]_0$/[has impact]$_a$ on CM.

REP3 varies one variable intrinsic to the object of study (i.e., the independent variable TSD). Accordingly, we study the following *null* $\mathbf{H}_0$ and alternative hypothesis $\mathbf{H}_0^a$:

- $\mathbf{H}\text{TSD}_0^a$: A motivated and cohesive TSD $[\textit{has no impact}]_0$/[has impact]$_a$ on the communication aspect.

7.3.7.1 Experiment Procedure

Before presenting the experiment procedure, we would like to highlight that we conducted several pilot studies, 2 in the university of Gothenburg, 1 in Aachen university, 1 in Lille university, and 1 in the Slovak university. To cover the treatments of our study, each pilot study involved 2 *Explainer-Receiver* pairs (B.Sc., M.Sc., or PhD students in SE). One pair was assigned to the G group using a graphical design description, and the second pair was assigned to the T group using a textual design description. These pilot studies helped us in:

- Designing a research protocol and assessing whether or not it is realistic and workable, especially in estimating the time that is required by: (i) the *Explainer* to understand the design (20 minutes), and (ii) the *Explainer* and *Receiver* to discuss the design (12 minutes).
- Identifying logistical problems and determining what resources (e.g, supervisors, software, and rooms) are needed for the actual experiments.
- Training the supervisors of the experiments.

The experiment procedure was created to define the process of the experiment and to ensure strict replications of the original experiment. Figure 7.4 presents the four main steps of the experimentation procedure:

- **Step 1**: To anonymize their identity and thus their answers, all participants were randomly assigned an identification number (ID). We asked the participants to bring their PCs to be able to answer the online pre- and post-task questionnaires. *Eduroam Internet* connection was available in the rooms where the experiments were running. Also, we asked the participants to bring a device to record the discussions (either by downloading audio-recording software on their PCs or by using a smart-phone with a recording application). We booked large university lecture-rooms which can host all *Explainer-Receiver* pairs with a sufficient distance between each pair. This helps to reduce voice interference to a minimum and produce better-quality audio recordings. We randomly assigned the participants to two groups (G and T). Furthermore, we randomly assigned each participant one role, *Explainer* or *Receiver*. After that, we asked the participants to answer the pre-task questionnaire.

- **Step 2**: Once all participants filled the pre-task questionnaire, the *Explainers* were taken to a second room where they received the design case specification and the design description (GSD or TSD). The *Explainers* were asked to understand the design that they received as good as they can in 20 minutes. During this time, the *Receivers* ensured that their recording software/devices were working as expected.

- **Step 3**: After 20 minutes, we took the design case specification from the *Explainers*, but let them keep the assigned design description (GSD or TSD). The *Explainers* and *Receivers* were randomly grouped in pairs in one or two rooms according to the number of participants and room's capacity. The pairs were then informed that, using the design descriptions (GSD or TSD), *Explainers* should explain the design to the *Receivers* in 12 minutes. We also informed the pairs that *Receivers* can ask clarification questions to the *Explainers*. When all participants were prepared, we asked the participants to start the audio recorder software and begin the discussions by introducing themselves (by mentioning the name/ID and role). This allowed us later to match the discussion records of the participants with their corresponding answers to the questionnaires.

- **Step 4**: After 12 minutes, the participants were informed that they should stop the audio recording. All documents, including *Receivers*' draft notes, were collected. Then, we asked all the participants to answer the post-task questionnaire individually. Lastly, we asked the participants to rename the audio recordings with their ID numbers and put the recordings in a USB flash drive that we provided.

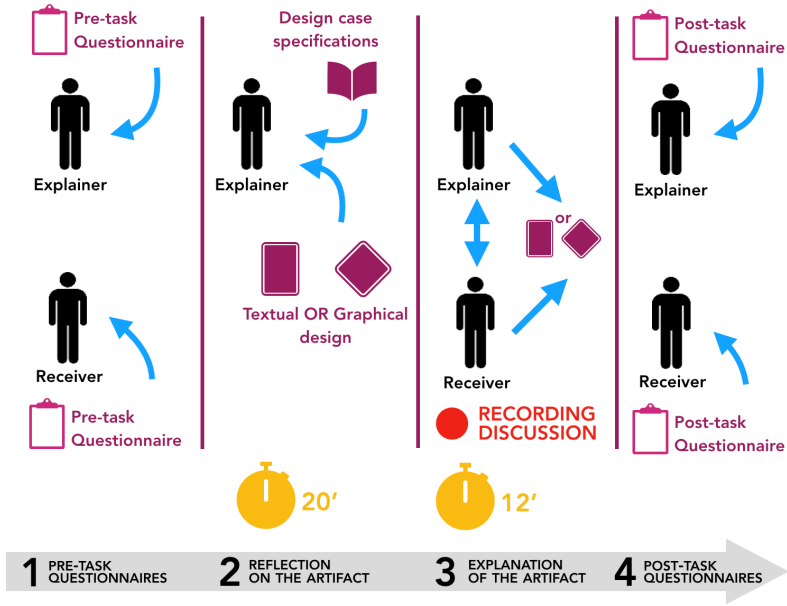


Figure 7.4: The five main steps of the experimentation procedure.

7.3.8 Data Analysis

The data of this study was collected via questionnaires and by audio-recording discussions between *Explainers* and *Receivers*. In this section, we describe three types of analysis procedures that we used:

- *Data Set Preparation*: To check and organize data collected from different sources and prepare it for analysis.
- *Descriptive Statistics*: To describe the basic features of the data by summarizing and showing measures in a meaningful way such that patterns might emerge from the data.
- *Hypothesis Testing*: To make statistical decisions by evaluating two mutually exclusive statements about a population and determining which statement is best supported by the sample data.
- *Meta-Analysis*: To obtain a global effect of a factor on a dependent variable by combining the effect size of different experiments, as well as assessing the consistency of the effect across the individual experiments [173].

7.3.8.1 Data Set Preparation

Data from 14 participants (7 pairs) were eliminated: 1 *Explainer-Receiver* pair from OExp as well as REP1, and 5 pairs from REP2. In particular, 5 pairs discussed the design assignment for too short time (less than 2 minutes) and decided to discuss other topics of their interest for the rest of the time. Moreover, the audio quality of the recorded discussion of 2 pairs was bad and the corresponding data from these pairs was eliminated. The final number of participants in each experiment is provided in Table 7.4.

Table 7.4: Final number of participants.

ExpID	Initial # of Subjects	Final # of Subjects	Eliminated Pairs	Reason of Elimination
OExp	50	48	1	- Bad quality of recorded discussion
REP1	36	34	1	- Too short discussion time (<2 minutes)
REP2	94	84	5	- Too short discussion time (<2 minutes) (4 cases) - Bad quality of recorded discussion (1 case)
REP3	60	60	0	N/A

The discussions between *Explainers* and *Receivers* were recorded by using either mobile phones or *Audacity*, an easy-to-use audio editor and recorder that works on multiple operative systems[9]. We transcribed approximately 23 hours of audio recordings and performed a manual coding of more than 2000 discussion records between *Explainers* and *Receivers*. For coding the discussions, we used the collaborative interpersonal problem-solving skill taxonomy of Margaret McManus and Robert Aiken [156], as presented in Figure 7.5. This taxonomy captures the collaborative interpersonal communication aspects; Active Discussion, Creative Conflict, and Conversation Management, which we described previously in Section 7.1. For instance, the following transcribed sentence: "*Can you explain why/how?*" is a *Request* for *Clarification* which contributes to *Active Discussion*. Another example: "*If... then*" refers to *Suppose*; one of the categories of *Argue* which contributes to a *Creative Conflict*. More examples are provided online[10].

NVivo[11] was used for coding the transcriptions. Prior to coding, we ensured coding/rating's reliability by performing two-way mixed Intraclass Correlation Coefficient (I-C-C) tests with 95% confidence interval on 9% of the data. In particular, three coders/rater were involved in measuring the I-C-C of the G group and T group of EXP, REP1, and REP2 (I-C-C value is 0.97 for group G and 0.96 for group T). Whereas, two coders/raters were involved for measuring the I-C-C of the G group and T group of REP3 (I-C-C value is 0.83 for group G and 0.92 for group T). The coding/rating reliability is positive. Indeed, according to [174], I-C-C is good when it is > 0.75 and ≤ 0.90 and excellent

[9] `https://www.audacityteam.org`

[10] `http://rodijolak.com/SE_Whispers/Problem_Solving_Skill_Taxonomy.pdf`

[11] `https://www.qsrinternational.com/nvivo`

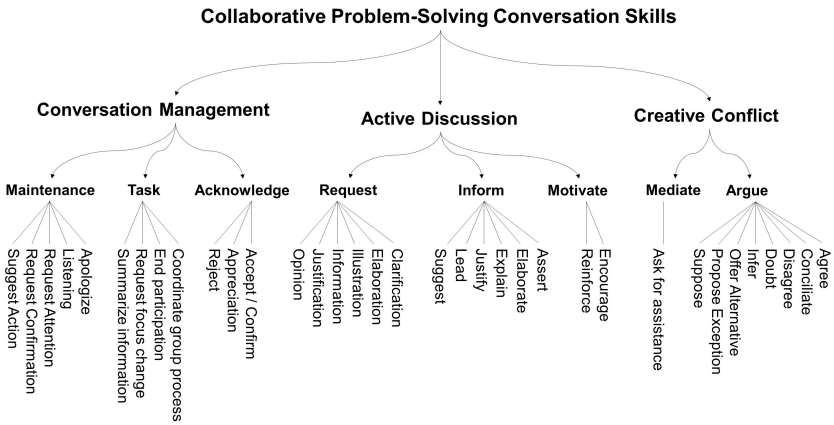


Figure 7.5: Collaborative interpersonal problem-solving conversation skills [156].

when it is > 0.90. Based on this result, the raters collaboratively continued to code the rest of the data i.e., 91% of the data.

7.3.8.2 Descriptive Statistics

By using IBM SPSS[12], we generated descriptive statistics, including *Box-plots* and *Mean +/- 1SD plots*, to analyze the collected data via questionnaires and audio recordings. In particular, we measured: means, medians, standard deviations, and ranges. These descriptive statistics help to analyze central tendencies and dispersion.

7.3.8.3 Hypotheses Testing

In the family of experiments, we wanted to compare two treatments/groups (G and T). So, we assigned our participants to these two groups by following the between-subjects design. In this setting, different people test each condition to reduce learning- and transfer-across-conditions effects. The collected data during the experiments include both interval and ordinal measures. Moreover, they are not normally distributed. Thus, we used non-parametric tests.

In particular, the hypotheses that we formulated in Section 7.3.7 seek to determine whether two independent samples have the same distribution. Therefore, these hypotheses were tested by performing the non-parametric independent-samples Mann-Whitney test.

[12] `https://www.ibm.com/analytics/spss-statistics-software`

7.3.8.4 Meta-Analysis

We perform a fixed-effect meta-analysis, as all factors that could influence the effect size are the same in all the experiments [173]. We use different scales to measure the communications aspects. Thus, for each experiment (i), we compute the effect size (G_i) by calculating the Hedges' g metric [175]. The assigned weight to each experiment is:

$$W_i = \frac{1}{V_{G_i}} \tag{7.1}$$

where, V_{G_i} is the within-experiment variance for the ith experiment. We obtain the global effect size by calculating the weighted mean M:

$$M = \frac{\sum_{i=1}^{k} W_i G_i}{\sum_{i=1}^{k} W_i} \tag{7.2}$$

According to [175], the effect size is small when $g \geq 0.2$; medium when $g \geq 0.5$; and large when $g \geq 0.8$. We report the result of the meta-analysis by using forest plots [173].

7.4 Results

In the first part of this section, we report the participants' perceptions of their design experiences and communication skills (the results of the pre-task questionnaire). After that, we present the results of the individual experiments and the performed meta-analysis. Finally, we report the participants' perceptions of their experience in working with different design representations (the results of the post-task questionnaire).

7.4.1 Perceived Design Experience and Communication Skills

Table 7.5 presents the questions of the pre-task questionnaire. It reports the id-number of the question, its description, and the experiment group.

The results to these questions are presented as box-plots in Figure 7.6. We report the id-number of the question and the experiment group on the x-axis. The y-axis presents a 5-point *Likert* scale, where 1 is the lowest score and 5 is highest score. We find that:

- the participants are somewhat familiar with software design (median = 3).
- the participants are familiar with software modeling and good in understanding and sense-making of UML models (median = 4).

- the participants are very good in reading, understanding, and sense-making of textual documentation (median = 5).
- the *Explainers* in the group G are neither poor or good in explaining their knowledge (median = 3), while the *Explainers* in the group T are good in explaining their knowledge (median = 4).
- the *Receivers* of the two groups (G and T) are good in building knowledge from conversing with others (median = 4).

Table 7.5: Pre-Task Questionnaire

Q	Description	G	Role
1	How familiar are you with software design?	G T	EXP/REC EXP/REC
2	How familiar are you with UML modeling?	G	EXP/REC
3	How good are you in understanding UML models?	G	EXP/REC
4	How good are you in sense-making of UML models?	G	EXP/REC
5	How good are you in reading a textual document?	T	EXP/REC
6	How good are you in understanding a textual document?	T	EXP/REC
7	How good are you in sense-making of a textual document?	T	EXP/REC
8	How good are you in explaining your knowledge to others?	G T	EXP EXP
9	How good are you in building knowledge from conversing with others?	G T	REC REC

Q: Question, **G**: Group

7.4.2 Individual Experiments

Table 7.6 shows the descriptive statistics of the studied communication aspects grouped by the two groups G and T. Considering *Explaining*, *Recall*, *Active Discussion*, and *Creative Conflict*, we observe that the unbiased estimate of the effect size, based on the standardized mean difference between the two groups (Hedges'g [175]), is positive. This means that there is a clear tendency in favor of using GSD.

For *Understanding*, the participants achieved better results when using TSD in OExp (negative g value). In contrast, the participants of REP1 and REP2 achieved better results when using GSD. For *Conversation Management*, the results show that the participants of all the experiments spent more effort on conversation management when using TSD.

We tested whether or not the distribution of the communication aspects (i.e., dependent variables) is the same across the two groups by running the Independent-Samples Mann-Whitney U Test. Table 7.7 shows the results of the test. The p-value is the probability of obtaining the observed results of a

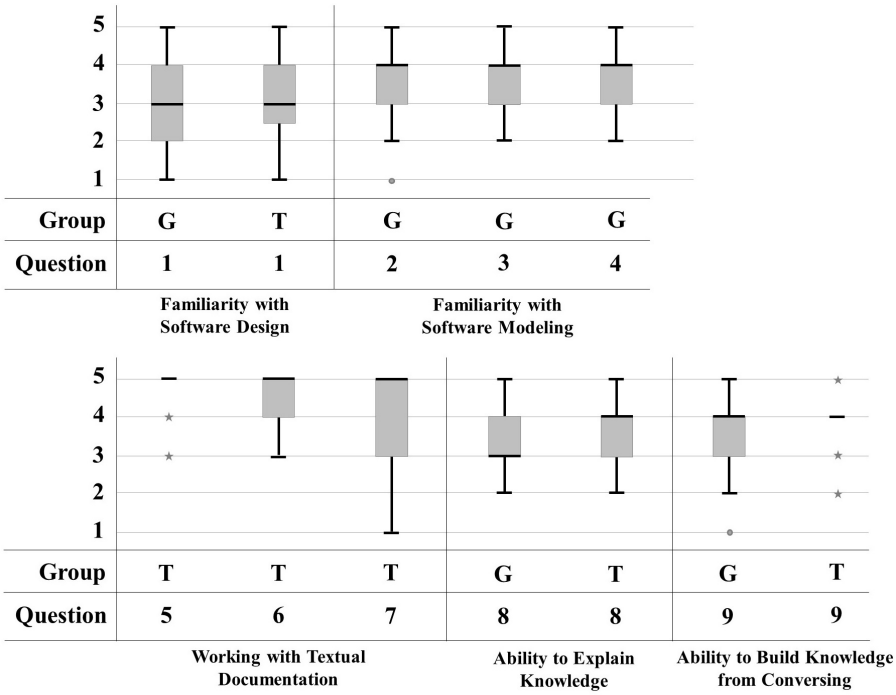


Figure 7.6: Results of the Pre-Task Questionnaire.

test, assuming that the null hypothesis is correct. We set the probability of type I error (i.e., α, probability of finding a significance where there is none) to 0.05.

The statistical power is the probability that a test will reject a null hypothesis when it is in fact false. As the power increases, the probability of making a type II error (β-value) decreases. A power value of 0,80 is considered as a standard for adequacy [176]. β-value is used to estimate the probability of accepting the null hypothesis when it is false. In REP1, we observe that there is a statistically significant difference in *Recall* between the two groups G and T (p-value = $0.037 < 0.05$, statistical power is 0.554). In REP2, we observe that there is a statistically significant difference in *Active Discussion* and in *Conversation Management* between the two studied groups (p-values = 0.010 and $0.011 < 0.05$, statistical powers are 0.694 and 0.705, respectively).

7.4.3 Meta-Analysis

In this section, we report and discuss the meta-analysis by means of forest plots. The squares in each forest plot indicate the effect size of each experiment.

Table 7.6: Descriptive statistics.

ID	Dept. Var.	Group G			Group T			Hedges' g*
		Mean	*Median*	*Std. Dev.*	*Mean*	*Median*	*Std. Dev.*	
OExp	EXP	3,864	4,000	0,710	3,654	4,000	0,689	0,295
	UND	1,545	1,750	0,754	1,808	2,000	0,906	-0,307
	REC	5,843	5,535	1,828	5,418	5,205	2,368	0,195
	AD	0,512	0,494	0,094	0,461	0,500	0,137	0,412
	CC	0,267	0,272	0,053	0,234	0,224	0,080	0,466
	CM	0,221	0,209	0,088	0,305	0,244	0,172	-0,580
REP1	EXP	3,900	4,000	0,553	3,563	4,000	1,209	0,365
	UND	1,875	2,000	0,741	1,625	1,750	0,806	0,317
	REC	6,973	7,165	1,747	5,598	6,125	1,981	0,725
	AD	0,457	0,447	0,107	0,409	0,411	0,063	0,505
	CC	0,166	0,189	0,067	0,118	0,121	0,053	0,745
	CM	0,377	0,406	0,124	0,473	0,488	0,085	-0,841
REP2	EXP	3,810	4,000	0,634	3,714	4,000	0,708	0,140
	UND	1,905	2,000	0,813	1,619	1,500	0,847	0,341
	REC	5,876	5,960	1,685	5,537	5,585	1,787	0,193
	AD	0,488	0,495	0,068	0,431	0,435	0,074	0,786
	CC	0,267	0,250	0,097	0,265	0,249	0,082	0,020
	CM	0,245	0,220	0,086	0,304	0,306	0,056	-0,793

* Hedges'g: unbiased estimate of the effect size based on the standardized mean difference [175].

The size of the squares represents the relative weight (squares are proportional in size to experiments' relative weight). The horizontal lines on the sides of each square represents the 95% confidence interval. The diamond shows the global effect size (the location of the diamond represents the effect size), while its width reflects the precision of the estimate (i.e., 95% confidence interval). The plot also shows the values of the effect size, weight, and p-value relative to each experiment and to the global measure. Positive values of the effect size indicate that the use of GSD increases/improves the effort/quality of the communication aspect, while negative values indicate that using TSD is the increasing/improving condition.

Figure 7.7 shows the forest plot for perceived quality of *Explaining*. We observe that the effect size values are positive. This implies that using a GSD has a positive effect on perceived Explaining quality. In other words, the participants' level of perceived explaining is better when using the GSD. However, despite this tendency, the global effect size is not statistically significant (p-value is $0.128 > 0.05$).

Observation 1 (Quality of Explaining).
We find that using a GSD has a positive effect on perceived Explaining *quality. However, there is no statistically significant difference in the perceived quality of explaining between the two groups (G and T).*

In a revised Bloom's taxonomy, Anderson et al. [95] outline a hierarchy of cognitive-learning levels ranging from remembering of a specific topic, over

Table 7.7: Independent variables Mann Whitney Test.

Exp-ID	Sample Size	Dependent Variable	Mann Whitney Test		
			p-value	*Statistical Power*	*β-value*
OExp	48	Explaining	0,367	0,168	0,832
		Understanding	0,300	0,179	0,821
		Recall	0,501	0,101	0,899
		Active Discussion	0,622	0,167	0,833
		Creative Conflict	0,140	0,198	0,802
		Conversation Mgt.	0,284	0,188	0,812
REP1	34	Explaining	0,519	0,184	0,816
		Understanding	0,342	0,151	0,849
		Recall	**0,037**	0,554	0,446
		Active Discussion	0,374	0,184	0,816
		Creative Conflict	0,110	0,550	0,450
		Conversation Mgt.	0,091	0,414	0,586
REP2	84	Explaining	0,636	0,097	0,903
		Understanding	0,152	0,332	0,668
		Recall	0,350	0,139	0,861
		Active Discussion	**0,010**	0,694	0,306
		Creative Conflict	0,990	0,050	0,950
		Conversation Mgt.	**0,011**	0,705	0,295

understanding and application of such knowledge, to advanced levels of analysis, evaluation, and creation. Figure 7.8 shows the hierarchy of the six cognitive learning levels. According to Anderson, *remember* is the recalling of the previously learned topic. *understand* is the ability to grasp the meaning of the topic by interpreting the knowledge and predicting future trends. *Apply* instead, comes on top of *understand*. It is the ability to use the acquired and comprehended knowledge in a new and concrete context or situation. In order

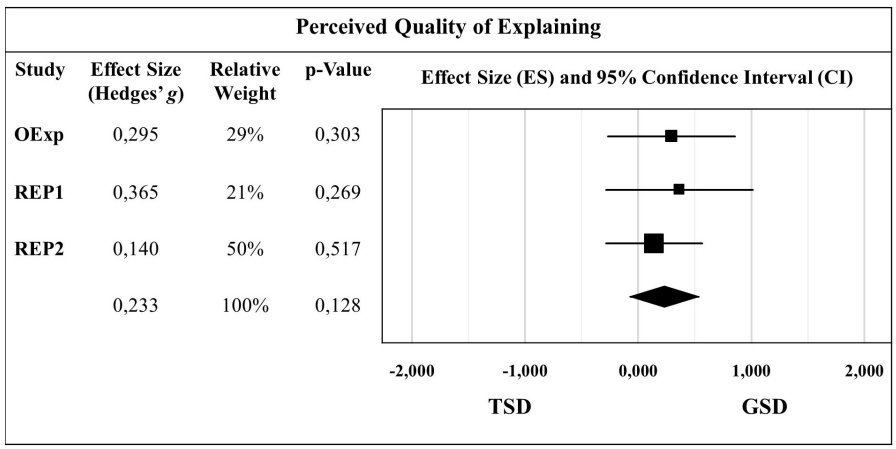


Figure 7.7: Meta-analysis for the perceived quality of Explaining.

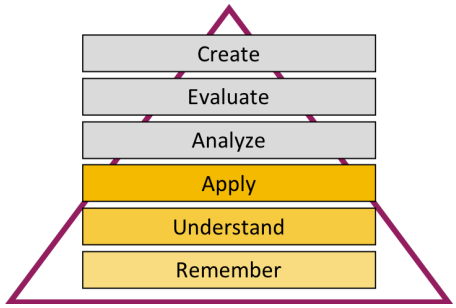


Figure 7.8: Bloom's taxonomy of cognitive learning.

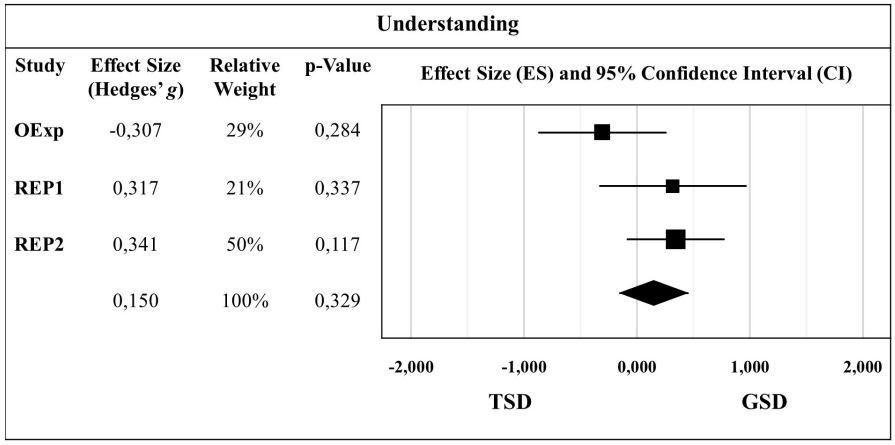
to measure the quality of understanding of our experiments' participants, we formulated three questions on design *maintenance* (these questions are provided in Section 7.3.6) which required the participants to use/apply their acquired knowledge in a new context (i.e., *apply* in Anderson's revised taxonomy).

The participants in the two groups (G and T) answered ten recall questions. We formulated the recall questions (see Section 7.3.6) to evaluate how well the participants do remember the design details after the discussions.

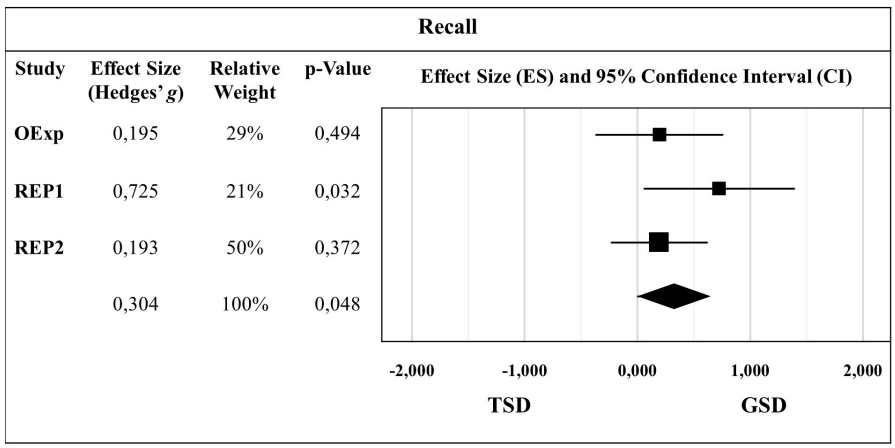
Figure 7.9 shows the the forest plot for quality of (a) *Understanding* and (b) *Recall* ability of design details. Regarding the quality of *Understanding*, the effect size value is negative for OExp, which means that TSD is the improving condition. For the other experiments (REP1 and REP2) the values of the effect size are positive. This implies that using a GSD in these two experiments has a positive effect on the understanding quality. Despite these tendencies, the global effect size is not statistically significant (p-value is 0.329). Considering the *Recall* ability, we observe that the effect size values are positive. This implies that using a GSD has a positive effect on the *Recall* ability. This effect is statistically significant and has a medium effect size for REP1. Furthermore, the global effect size is statistically significant (p-value = $0.048 < 0.05$).

Observation 2 (Quality of Understanding).
In OExp, we find that using a TSD has an advantage in improving the Understanding. *Whereas in REP1 and REP2, we find that using a GSD is the improving condition. Globally, there is no statistically significant difference in the quality of understanding between the two groups.*

Observation 3 (Recall Ability).
We find that using a GSD has a positive, statistically significant effect on Recall. *This suggests that using a GSD during design communication has an advantage in improving the recall of design details.*



(a)



(b)

Figure 7.9: Meta-analysis for quality of Understanding (a) and Recall (b).

Figure 7.10 shows the the forest plot for collaborative interpersonal communication dimensions: *Active Discussion* (AD), *Creative Conflict* (CC), and *Conversation Management* (CM). Considering AD and CC, we observe that the effect size values are positive. This implies that using a GSD has a positive effect on the amount of ADs and CCs. The global effect size for AD is statistically significant (p-value = $0.005 < 0.05$). However, the global effect size for CC is not statistically significant (p-value = 0.162).

Considering CM, we observe that the effect size values are negative. This implies that the effort on CM is bigger when using TSD. The global effect size

for CM is medium and statistically significant (p-value = 0.001 < 0.05).

Observation 4 (Active Discussion, AD).
We find that a GSD fosters more AD than a TSD. This suggests that GSD' users question, inform, and motivate each other more than TSD' users.

Observation 5 (Creative Conflict, CC).
We find that using a GSD has a positive effect on the amount of CC discussions. However, this effect is not statistically significant between the two groups (G and T).

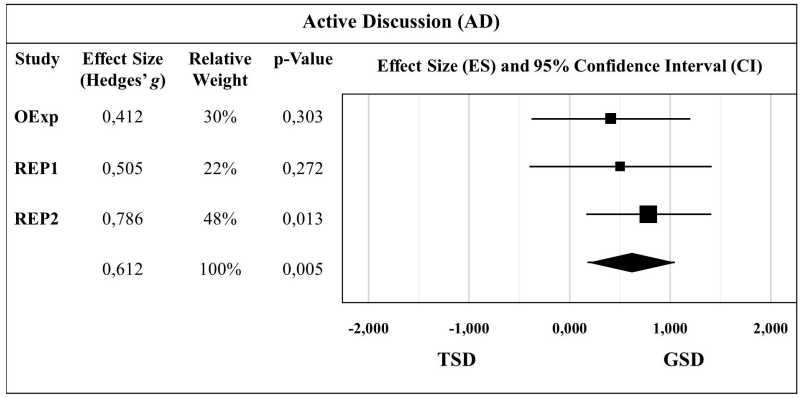
Observation 6 (Conversation Management, CM).
Using a GSD requires less CM effort than using TSD. The effect is statistically significant. This suggests that GSD' users do less coordination and acknowledgment of communicated information than TSD' users.

7.4.4 Motivated and Cohesive TSD

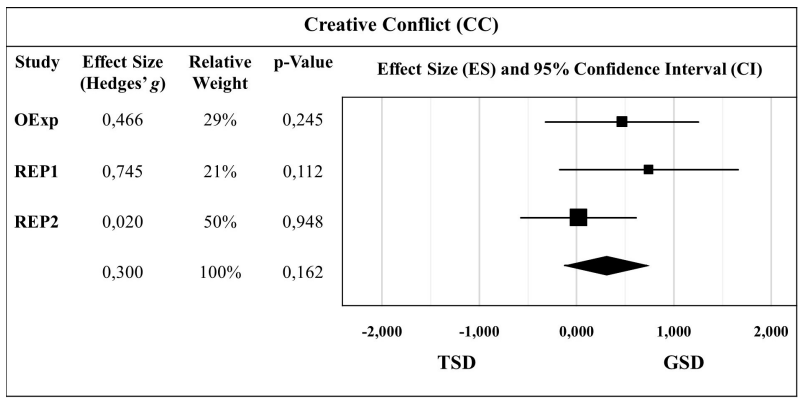
Falessi et al. [177] suggested that documentation of software design rationale could support many software development activities, including analysis and re-design. Tang et al. [178] conducted a survey of practitioners to probe their perception of the value of software design rationale and how they use and document it. They found that practitioners recognize the importance of documenting design rationale for reasoning about their design choices and supporting the subsequent implementation and maintenance of systems.

The goal of running REP3 is to know how a *motivated* and *cohesive* TSD (as described previously in Section 7.3.5.1 – Altered TSD) could influence design communications. To this end, we used a different TSD in REP3, which includes a rationale that motivates why the MVC paradigm is selected for structuring the design. Moreover, we organized the information/knowledge in the TSD and made it cohesive. In particular, the relationships of each entity are reported right after describing it, instead of being reported with all the other relationships in the 'relationship section' at the end of the TSD.

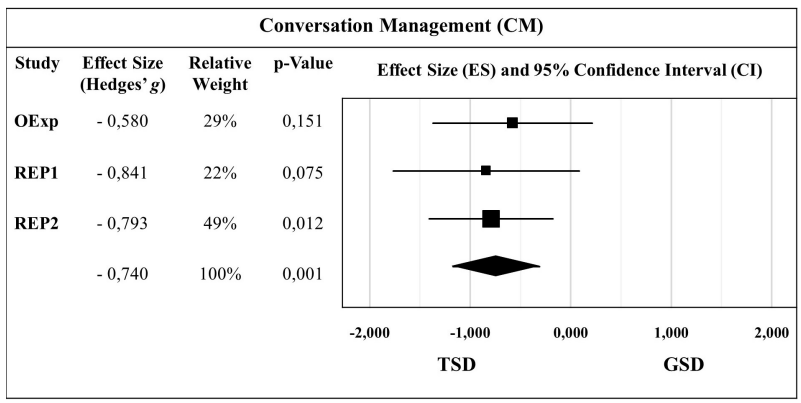
To achieve the goal of REP3, we compared the communication aspects: *understanding*, *explaining*, *recall*, and *interpersonal communication* of the text groups of OEpx, REP1, and REP2 (*Text1*) with the text group of REP3 (*Text2*). The results of comparing Text1 with Text2 are presented in Table 7.8. We performed the independent-samples Mann-Whitney U test of *explaining*, *understanding*, and *recall* as for these three indicators, the number of observations in *Text2* is ≥ 30. The p-value of *explaining* is 0,049 (< 0,05) and the effect size (Hedges' g) is negative (i.e., using TEXT2 is the improving condition).



(a)



(b)



(c)

Figure 7.10: Meta-analysis for interpersonal communication: Active Discussion (a), Creative Conflict (b), and Conversation Management (c).

Moreover, the p-value of *recall* is 0,008 (< 0,05) and the effect size is negative and medium. We can state that the distribution of *explaining* and *recall* are not the same across the two groups (Text1 and Text2).

Table 7.8: Descriptive statistics and independent-samples Mann-Whitney U test of difference between the two groups: Text1 and Text2.

Dept. Var.	Text 1				Text 2			
	Mean	*Median*	*Std.Dev.*	*#Obs*	*Mean*	*Median*	*Std.Dev.*	*#Obs*
EXP	3,667	4,000	0,812	84	3,967	4,000	0,964	30
UND	1,679	1,500	0,853	84	1,800	2,000	0,726	30
REC	5,512	5,585	1,995	84	6,664	7,080	2,226	30
AD	0,436	0,439	0,096	42	0,625	0,627	0,110	15
CC	0,228	0,224	0,093	42	0,113	0,118	0,058	15
CM	0,336	0,306	0,126	42	0,262	0,255	0,114	15

Dept. Var.	Mann Whitney Test			
	p-value	*Hedges'g*	*Power*	*β-value*
EXP	**0,049**	-0,349	0,360	0,640
UND	0,534	-0,147	0,103	0,897
REC	**0,008**	-0,556	0,722	0,278
AD	*small sample size*			
CC	*small sample size*			
CM	*small sample size*			

* Effect Size (Hedges'g [179]) is small when $g \geq 0{,}2$; medium $\geq 0{,}5$; large $\geq 0{,}8$

Observation 7 (Cohesive and Motivated TSD)
We suggest that a cohesive *TSD that* motivates *the design choices with rationale can enhance the perceived* explaining *(during design communication) and* recall *ability of design details.*

7.4.5 Perceived Experience in Working with Different Design Representations

Table 7.9 presents the questions of the post-task questionnaire. It reports the id-number of the question, its description, and the experiment group.

The results to these questions are presented as box-plots in Figure 7.11. We report the id-number of the question and the experiment group on the x-axis. The y-axis presents a 5-point *Likert* scale, where -2 is the lowest score and $+2$ is highest score. The results indicate that:

- the participants perceive that the design is very clear (median = +2).
- the participants perceive that they are good in recalling: a conversation, UML design models, and textual documentation (median = +1).

- the participants in the group G perceive that models are helpful in understanding the design (median = +1).
- the participants in the group T perceive that having diagrams would have helped in understanding the design (median = +1).
- the *Explainers* in the group G perceive that models are very helpful in explaining the design (median = +2).
- the *Explainers* in the group T perceive that having diagrams would have helped in understanding the design. (median = +1).

Table 7.9: Post-Task Questionnaire

Q	Description	G	Role
1	How clear was the design of the system that you received?	G	EXP/REC
		T	EXP/REC
2	How good are you in recalling (remembering) a conversation?	G	EXP/REC
		T	EXP/REC
3	How good are you in recalling (remembering) UML design models?	G	EXP/REC
4	How good are you in recalling (remembering) a textual document?	T	EXP/REC
5	Did the diagrams help you in understanding the design?	G	EXP/REC
6	Do you think having diagrams would have helped in understanding?	T	EXP/REC
7	Did the diagrams help you in explaining the design?	G	EXP
8	Do you think having diagrams would have helped you in explaining?	T	EXP

Q: Question, **G**: Group

7.5 Discussion

Our experiments investigate whether design communication between software engineers can become more effective when using GSD instead of TSD to exchange design information. To this end, we investigate whether using a GSD affects six considered communication aspects (Understanding, Explaining, Recall, Active Discussion, Creative Conflict, and Conversation Management) differently from using a TSD (R.Q.1). Moreover, we study whether a *cohesive* and *motivated* TSD improves design communication (R.Q.2).

The global effect size of the perceived explaining quality is positive. This means that using a GSD has a positive effect on the perceived explaining quality. Similarly, the global effect size of the understanding (i.e., maintenance task) score is positive, which means that the score of the GSD users is better than the score of TSD users. Nevertheless, by considering distributions of the scores we neither find a statistically significance difference in the quality of explaining (Observation 1) nor in the quality of understanding (Observation 2) between the two groups: G and T.

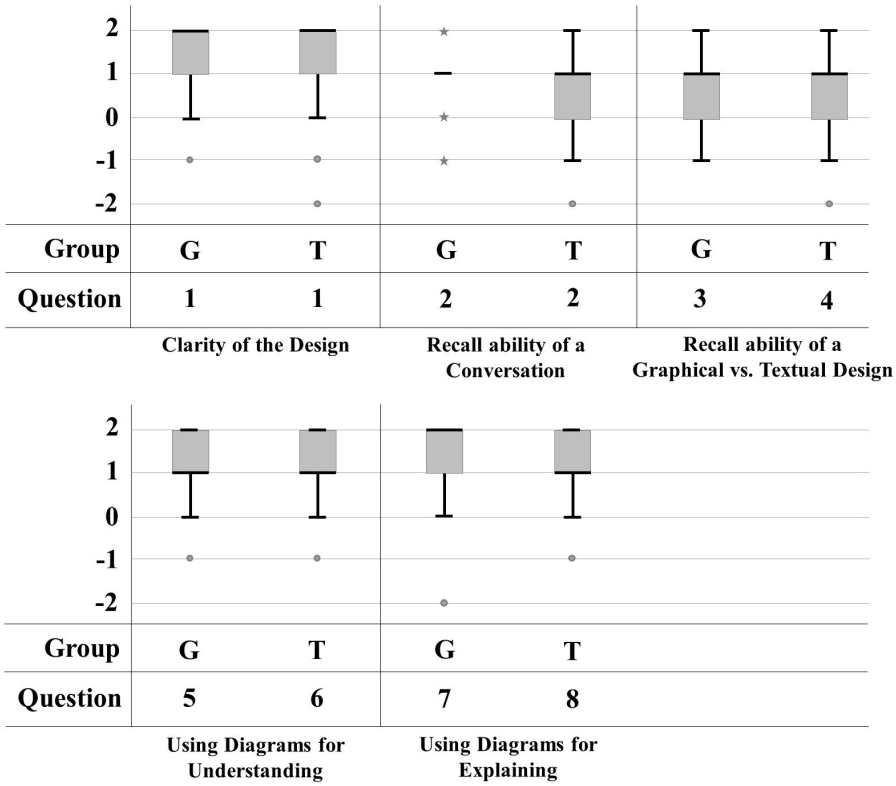


Figure 7.11: Results of the Post-Task Questionnaire

While analyzing the recorded, and further transcribed, discussions between the *Explainers* and *Receivers*, we interestingly observed a difference in the explaining approach between the *Explainers* of the two groups. Figure 7.12 provides an illustration of the observed explaining approaches in the two groups. On the one hand, the *Explainers* of a TSD tended to explain the three modules of the MVC sequentially: Firstly the model entities, then the controllers, and lastly the views, as these modules are orderly presented in the textual document. We think that this trend is intrinsically imposed by the nature of textual descriptions where the knowledge is presented sequentially on a number of consecutive ordered pages. On the other hand, the *Explainers* of the GSD had more freedom in explaining the design. Indeed according to their explaining preferences, the *Explainers* of the GSD tended to jump back and forth between the three MVC modules when explaining the design. Based on this, we suggest that a GSD has an advantage over the TSD in unleashing *Explainers*' expressiveness when explaining the design, as well as in helping

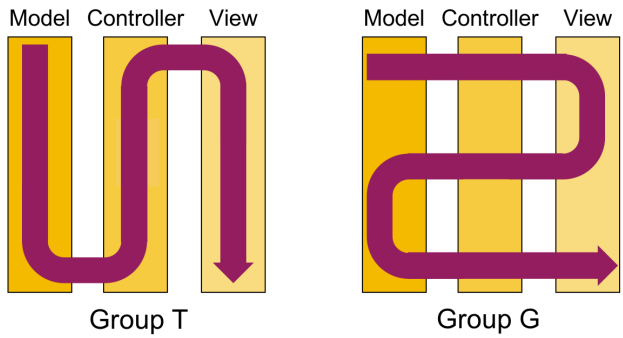


Figure 7.12: Observed explaining approaches used in group G and group T.

navigation and getting a better overview of the design. However, developers might not have this advantage when explaining many GSDs (e.g., many UML diagram) spread on different pages within a software design documentation.

We found that using a GSD is better than a TSD for recalling the details of the discussed design (Observation 3). This is actually inline with Maede et al. [168], who suggest that drawing graphical notations brings more recall benefits than writing textual words in younger and older adults.

Graphical representations are considered better than the textual in representing information which deals with relationships between entities [180]. One of the recall questions that we used to measure the recall ability of the participants is concerned with the relationships between the entities of the software architecture design. We compared the score (interval variable; min is 0 and max is 1 point) of the two groups on this question. On average, the users of the graphical representation were slightly better in recalling the relationships between the entities (G: Mean= 0.506; Std. Dev.= 0.331) vs. (T: Mean= 0.423; Std. Dev.= 0.347). However, this difference is not statistically significant (Sig.= 0.128 $>$ 0.05; Hedges' g= 0.244; Power= 0.338).

The *Chinese Whispers* game is often invoked as a metaphor for miscommunication. In this game, the first player often fail to recall all the information of the initial message that she/he receives. Likewise, the second player often fail to recall all the information of the message that she/he receives from the first player, and so on for the rest of the players. In the same manner, the *Explainers* in our experiments failed to *recall* all the design details that we asked for in the post-task questionnaire (Mean Score= 3.319; Std. Dev.= 0.855). The Receivers were, as expected, worse than the *Explainers* in recalling the design details (Mean Score= 2.492; Std. Dev.= 0.885). Moreover, we found that the difference in *recall* ability betweeen *Explainers* and *Receivers* is statistically significant (Sig.= 0.000 $<$ 0.05; Hedges' d= 0.946, Power= 0.999).

Based on empirical results, we find that a GSD fosters more *Active Discus-*

sion (AD) than TSD (Observation 4), while reducing *Conversation Management* (CM) at the same time (Observation 6). In the skill taxonomy of McManus and Aiken [156], the communication activities in the AD category generally aim at helping an active exploration of the discussed argument by encouraging information requesting, clarification, or elaboration. In contrast, the branch of CM comprises communication activities that generally contribute less to active information requesting or clarification, such as acknowledging or coordinating group tasks. Consequently, we suggest that using a GSD as a basis for software design communication promotes an active exploration of the communicated designs, which in turn helps to improve the effectiveness of software design communication.

There is no significance difference in Creative Conflict (CC) discussions between the two groups (Observation 5). We suggest that the type of design description does not influence design argumentation and reasoning. Alternatively, we think that the context, complexity of the design, available knowledge, or the application of reasoning techniques might affect the quality of design argumentation and reasoning discussions, as suggested by Tang et al. [99].

It is widely assumed that model-based techniques support communicating software [181]. Our findings support such assumption and prove that using a GSD (i.e., a software model) improves the recall ability of the discussed design details, fosters *Active Discussion*, and at the same time reduces less useful conversation on activities management.

We conducted REP3 to better calibrate our findings of the differences between GDM and TSD. We found that a motivated (i.e., augmented with rationale) and cohesive TSD helps to improve both the explaining and recall ability of the discussed design details (Observation 7). This finding is indeed inline with Tang et al. [60] who stated that discussing the reasons of making software design choices (i.e. design rationale) positively contributes to the effectiveness of software design discussions by facilitating communication and design knowledge transfer.

7.5.1 Threats to Validity

Our family of experiments is subject to threats to their construct validity, internal validity (causality), external validity (generalizability), and conclusion validity. We highlight these issues and discuss related study design decisions.

7.5.1.1 Construct Validity

Constructs validity refers to how well operational measures represent what researchers intended them to represent in the study in question. In this study, we used a single method for measuring the impact of different design representation per each communication aspect. To mitigate this issue, we did not only rely on questionnaires, but also recorded, transcribed, and later evaluated

the communication observed during the experiments. Nonetheless, leveraging additional methods to probe the explaining, understanding, recall, and interpersonal communication skills of the participants might help to better investigate the effects of different design representations. Such methods, for instance, might comprise conducting actual software design or software engineering tasks after receiving the explanation. However, this would introduce a multitude of other variables (e.g., the programming language or IDE used) that either can be hardly controlled or demand for drastic simplification, thus reducing our experiments' generalizability.

Another threat to construct validity could arise from discretizing the measurement of continuous properties, such as the participants' familiarity with software design or their expertise with UML. This challenge has been investigated for balanced *Likert* and identified as not comprising generalizability [182].

7.5.1.2 Internal Validity

The questionnaires to evaluate the participants' performance raise threats to internal validity themselves: For instance, the participants might interpret the Likert scales we have used differently, might have avoided extreme responses (central tendency bias), and - as the participants evaluated their communication skills themselves - might be biased towards overestimating or underestimating their skills, which might be subject to different effects on their introspection. To support comprehension and reproduction of results, we use established surveys where possible and provide all materials on the experiments' companion website. Nonetheless, completely mitigating the potential effects of surveys' general deficiency requires the development of novel methods to test familiarity and understanding of UML designs and textual designs, as well as communication skills. While for the latter, specifically tailored exercises might be feasible to evaluate the skill level, conducting these, (a) requires unbiased instruments as well and (b) might affect our experiments. A specific challenge of our family of experiments regarding the questionnaires arises from conducting the REP2 survey in French, whereas the other experiments used English documents. While this generally could affect the results, the experimenters of REP2 had the task documents and questionnaires professionally translated and reviewed to maintain the consistency of the communicated information.

To mitigate the effect of limited preparation and explanation time – the *Explainers* had 20 minutes to understand the design and 12 to discuss it with the *Receivers* – we conducted multiple pilot studies at all sites prior to the actual experiments to understand how much time is required. After running the pilot studies, we increased the initially considered 10 minutes of discussion to 12 based on the feedback of the participants of the pilot studies. Afterwards, we conducted another pilot study that confirmed that both times are considered suitable for the tasks.

Other challenges to internal validity stem from the selection of our experi-

ments' participants. Potential confounding factors include that due to randomly assigning the participants to the G or T group, certain personality types are prevalent in one of the groups – which could affect results. By measuring the Big Five factors of personality [183], we checked that this is not the case: the distribution of the five personality factors (Extraversion, Agreeableness, Conscientiousness, Neuroticism, and Openness) is the same across the two groups. Similarly, it could have affected our findings that the members of one of the two groups have significantly more experience with software design than the members of the other group. The pre-task questionnaire establishes that this is not a problem of our study. Other issues could have arisen from our participants being unfamiliar with UML designs or textual designs but the pre-task questionnaire shows that this is not the case. We assume that this is due to the participants' educational backgrounds (in which processing textual designs for exercises or exams is common).

The textual representations used in this research are structured by indentation, indexing, and grouping information, which are helpful for information retrieval [184]. However, these might have positively affected the quality of TSD communication. Similarly, the MVC entities in the graphical representation were highlighted by colors, which is also helpful for information retrieval [184]. This might have also positively affected the quality of GSD communication. If the descriptions of the entities in TSD were tangled and if the entities of the GSD were not colored, then the quality of communication of these two representations might have been different and less efficient. As this is the case in both groups (i.e., G and T), we assume that this should not affect the results of the comparison too much. This, however, raises threats to validity: The augmentations to the textual representation might yield other (stronger or weaker) effects than the class diagram coloring. As both, coloring in graphical models and structuring of textual design documents, is common in industrial practice, we do not consider this a significant threat over using unstructured text and uncolored diagrams.

Some *Receivers* of the text group were drawing (informal) class diagrams while being explained to. Hence, there might be an interaction of both treatments, but with only six (2.5%) of the *Receivers* being affected, the effect of this combination of both representations is negligible.

Another threat might arise from the using textual survey questions as method to investigate the benefits of textual and graphical designs. Maybe, textual design representations yielded better answers to the questions because they are syntactically closer than graphical designs to the textual answers. This threat could be mitigated through leveraging graphical questions and answers in the surveys. While this would be feasible for the answers, for formulating the questions as graphical class diagrams, this would entail a new syntax which might yield further threats.

7.5.1.3 External Validity

Threats to external validity indicate to which extent the results of our study can be generalized. Due to working at software engineering research and education institutes, we selected students with strong software engineering backgrounds of our Universities. While this prevents generalizing results to software developers with different backgrounds (e.g., developers in computer vision, artificial intelligence, or robotics), software design aims at software architecture from which we expect strong software engineering backgrounds.

Also, we conducted our studies with students instead of software design practitioners. Hence, the participants involved in our experiments may not represent the general professional population of software engineering practitioners. While this limits us from generalizing our findings to other subjects (i.e., domain experts, professional software architects, industrial practitioners in the field), the differences between students and professional software developers in performing small tasks are generally very small [185]. We, therefore, consider our findings as a basis to extend our study to a larger community of software engineering practitioners.

Another threatening effect is that the population of professional software developers yield a larger age range than students. With recall abilities changing over time [186], this limits generalization of our results to professional software developers of the same age range – between 20 and 30 years – than software engineering students and PhD students (as proposed in [187]).

Moreover, the studies were conducted in educational contexts, i.e., contexts in which the students usually are evaluated and graded. This generally might have improved their performance (Hawthorne effect). However, as this applies to both groups, this does not affect our results.

Due to the outline of our experiments as single one-hour sessions and their popular context in sports that are easily relatable, we can exclude threats regarding history or maturation. The participants could neither have been effected from previous events of the experiment as there have not been any.

Moreover, as we used the same two textual/graphical notations in all experiments, this limits generalizability of our results to other textual or graphical representations, i.e., differently structured text or differently highlighted class diagrams. This, however, is a threat independent of the specific choice of representation and demands for studies deploying multiple (popular) representations – which demands correctly identifying industrially relevant forms of representation and yields further threats to generalizability.

Another challenge to generalizability might arise from the constructs investigated, i.e., whether structured textual design documents and colored UML class diagrams actually are relevant to communicating design decisions in industry. While the use of UML in software design and engineering is undaunted in various domains (cf. [17, 188]), so is the use of textual documents to describe software designs [189–191]. However, using a specific form of structured text

for communicating design decisions limits generalizability to this form of text. For instance, in requirements engineering, there are different tools that support capturing textual requirements and design decisions using different textual representations [192] and using these might entail different effects.

Generalizability might also be challenged by the size of documents used of investigation. There are no studies on the number of classes per class diagram in industrial software engineering projects. However, a report on numbers of classes per class diagram used in different lectures reports that in 101 diagrams from 5 different courses, the maximum number of classes per diagram is 40, with the minimum being 3 and the average being 10.75 [193]. This might indicate that our design class diagram of 28 classes is a bit more complex than it would be usual for education (and hence be more realistic regarding industrial challenges). Another study investigated 100 android applications from open-source repositories [194]. Here, only the average size of these applications as 90 classes is reported. While this does not report how these would be aligned in different class diagrams, assuming the these cover at least three different concerns (e.g., model, view, and controller) appears reasonable, which would entail 30 classes per class diagram on average and would be in line with the 28 classes presented in our experiment. Therefore, we consider the size of the experiments' class diagrams relevant. For the textual design documents, we are unaware of any studies on their average size, but due to them containing the same information as the class diagrams, which are of relevant size, we conclude that these should be as well. However, this needs further investigation and might challenge the generalizability of our results. Also, the effect of the number of classes conveyed in both representations might effect understanding and recall. This also demands for further investigation.

Similar to the threat of using specifically indented and colored documents, the optimality of their representations might challenge generalizability of our results as it might be conceivable that there are better suitable textual or graphical representations that lead to different results. To the best of our knowledge, the best representations of textual design documents and graphical class diagrams still have to be identified and whether these are optimal for any domain needs to be investigated. Nonetheless, differently presented textual or graphical designs might have yielded different effects. This, however, is a threat to generalizability that holds for any study investigating a finite number of alternative treatments where infinitely many are possible and needs to be considered when applying our results.

Also, the experimental conditions (scope, team size, duration, etc.,) might differ from real-world conditions and limit generalizability of results. Nonetheless, especially in the use case of onboarding of job newcomers by experienced developers and designers, this challenge is of practical interest as indicated by Ericsson's "*Experience Engine*" initiative (cf. section 7.1).

7.5.1.4 Conclusion Validity

Threats to conclusion validity challenge how reasonable a research or experimental conclusion is. In our study, these threats might arise, mainly, through concluding the existence of in-existing differences (type I error) and concluding the in-existence of existing differences (type II error).

We conducted hypotheses testing to determine whether two independent varaibles have the same distribution. We might have committed type I error and incorrectly rejected the null hypothesis (false positive), or committed type II error and incorrectly accepted the null hypothesis (false negative). However, we considered the *significance* and minimized the risk of detecting a non-real effect by setting the α value to 0,05. Also, we analyzed the *sensitivity* by discussing the effect size and statistical power of our tests.

7.5.2 Implications

Using GSD to communicate software designs produces *more active discussion, less conversation management, and better recall.* These effects contribute to deepening the active exploration of the discussed design [195], which is why we consider using GSD beneficial to communicating software designs. While for identification of design errors, textual descriptions seem to be more efficient [29] than GSD. Our findings suggest the use of GSD as a basis for communicating designs with the objective of transferring design knowledge, which is in line with the observed benefits of graphical documents on recall [168].

Our findings, however, assume that the textual design document accurately represents the GSD. Often, however, these natural language documents yield ambiguities or omit details that can be missed less easily in graphical descriptions. We assume that this can be due to graphical descriptions, such as UML class diagrams, being accessible for model checking to identify, e.g., missing associations or missing types. Future work should investigate whether textual artifacts used in practice indeed represent the underlying design accurately.

With REP3, we also investigated the effects of a cohesive and motivated TSD on design understanding, explaining, recall, and interpersonal communication. As we found a difference in explaining and recalling ability between both groups: original TSD and altered TSD (Observation 8), future research in improving software design communication should investigate comparing benefits of augmenting GSD with textual motivation and rationale as well.

7.5.3 Generalization

Generally, we found that communicating design with a GSD yields better discussions and better recall. We believe that these effects are not limited to software design documents but transfer to graphical software descriptions

in general. While, for instance, UML class diagrams meant for implementation might differ in the level of detail, but not in the general representation. Applying our findings regarding the benefits of (i) GSD over TSD and (ii) cohesive TSD with rationale to other kinds of software artifacts can yield benefits for their communication and consumption as well. For instance, as requirements documents become more complex [162], augmenting these with graphical representations or rational could, ultimately, improve requirements engineering. Model-based systems engineering [196] traditionally considers graphical representations. Nonetheless, similar improvements could be achieved as the collaborating stakeholders from various domains could benefit from being provided rationale of design decisions made in other domains.

There also is research in textual modeling [197], which leverages textual models with well-defined semantics for software design and development. As such, these textual models are in-between GSD and TSD and whether our results translate to textual software models, such as UML/P class diagrams [198], needs further investigation.

Similarly, the observed benefits of GSD are subject to the viewpoint we selected in a fashion that allows presenting the complete design description (i.e., model) on a single sheet of paper. For more complex diagrams, this might not scale-up. However, we assume that the textual design document (currently three sheets of paper) scales-up even worse. Consequently, we believe that the effects of software design representation on large designs with hundreds or thousands of elements will be even more prominent.

7.6 Conclusion and Future Work

We conducted a family of experiments to study the effect of using graphical versus textual software design descriptions on software design communication. According to [94–96], we considered the following communication aspects:

- *Explaining*: communicating intellectual capital from one person to others.
- *Understanding*: receiving others' intellectual capital.
- *Recall*: recognizing or recalling knowledge from memory to produce or retrieve previously learned information.
- Collaborative Interpersonal Communication, which includes:
  - *Active Discussion*: questioning, informing, and motivating others.
  - *Creative Conflict*: arguing and reasoning about others discussions.
  - *Conversation Management*: coordinating and acknowledging communicated information.

Based on empirical findings, we suggest that a *graphical* software design descriptions (GSD) improves design-knowledge transfer and communication by:

- promoting *Active Discussion* between developers,
- reducing *Conversation Management* effort, and
- improving the *Recall* ability of design details.

Furthermore, we found that *motivating* (by adding design rationale) and making textual design descriptions *cohesive* (by organizing the design knowledge in the document) help to enhance the explaining and recall of their details.

7.6.1 Impacts on Practitioners

In a field study of the Software Design process, Curtis et al. [199] identified broad communication and knowledge sharing as two factors that have effects on software quality and productivity. According to our findings, we suggest that the use of GSD can help in improving design-knowledge sharing and communication. Hence, we identify the following impacts on practitioners:

- *Agile Practices.* Agile development practices include several processes in which communication is at least involved, if not central [200]. Daily meetings are, by definition, the perfect example of agile ceremony which completely relies on communication. According to Karlström et al. [201], holding daily meetings as a mechanism for design problem solving appeared to have positive effects on the communication of the design issues. Based on our findings, introducing GSDs in daily discussions about design decisions would enhance the communication quality between participants, which in turn could strengthen the impact of applying agile practices in software engineering projects.

- *Reducing Development Efforts.* Multiple studies demonstrated that communication is one of the most time-consuming tasks in software development, requiring more effort than any other development activity [27] and taking up to two hours a day per each individual developer [202]. As face-to-face communications are strongly preferred when possible [171, 202], the use of GSD as a support for design-related communication could be of benefit for productivity. Minimizing the required effort for communication would provide developers with more time at disposal as well as reduce developers mental-load, so they can focus on different tasks.

- *Satisfaction and Productivity.* Although there is no notable difference in the perceived quality of explaining between group G and group T, all participants from the two groups reported that a GSD indeed helped, or

would have helped them, in explaining the design. Accordingly, we think that using GSDs would make the communication of the design easier and increase the satisfaction of developers. Graziotin et al. [203] reported that satisfaction is directly correlated to productivity. So, we suggest the use of GSD in design meetings in order to increase the productivity of software development teams.

- *Pedagogical medium.* By observing of the explaining approaches in the two groups, we suggested that a GSD has an advantage over the TSD in helping navigation and getting a better overview of the design. Even though this requires more investigation, we suggest that, due to its nature, a GSD provides more adaptability and extra degrees of *explaining* freedom, which makes it a better pedagogical medium for face-to-face design knowledge transfer.

- *Design Rationale.* Falessi et al. [177] state that documenting design rationale could support many software development activities, such as an impact analysis or major redesign. Tang et al. [204] find that design reasoning (i.e., discussing rationale) improves the quality of software design. In this paper, we find that a TSD that *motivates* the design choices with rationale can enhance the *recall* and *explaining* of its design details. Accordingly, we suggest the producers of software design tools (graphical or textual) to provide explicit mechanisms for capturing and retrieving design rationale. Furthermore, we encourage developers to include design rationale in design documentations to improve design communication, which in turn should improve the overall communication and collaboration, and thus the productivity, in SE projects.

7.6.2 Future Work

One future direction is to replicate the experiment in order to address and minimize the threats to the validity of our research design and results. For instance, by replicating the experiment with a more complex graphical or textual software design description, by changing the order of complexity of the recall and maintenance tasks, or by involving professionals. Moreover, to maximize the benefits, another line of research is to investigate new techniques or approaches that would enhance the effectiveness of software design communication. One example of these approaches is proposed in a study by Tang et al. [99] where a reminder card approach was employed to improve software design reasoning discussions. Another example is proposed by Robillard et al. [100] who argue that automatic on-demand documentation generators would effectively support the information needs of developers.

acknowledgments We would like to thank Prof. Robert Feldt for his valuable suggestions and inspiring discussions about this work.

Ethical Issues In this study, we considered the major ethical issues according to [205]: informed consent, beneficence– do not harm, and respect for anonymity and confidentiality.

Bibliography

[1] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis Lectures on Software Engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[2] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.

[3] B. Selic, "What will it take? a view on adoption of model-based methods in practice," *Software & Systems Modeling*, vol. 11, no. 4, pp. 513–526, 2012.

[4] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice," *Software & Systems Modeling*, pp. 1–23, 2016.

[5] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proceedings of the 33rd Int. Conf. on Software Engineering*. ACM, 2011, pp. 471–480.

[6] P. Devanbu, T. Zimmermann, and C. Bird, "Belief & evidence in empirical software engineering," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 108–119.

[7] P. Ralph and Y. Wand, "A proposal for a formal definition of the design concept," *Design requirements engineering: A ten-year perspective*, vol. 14, pp. 103–136, 2009.

[8] R. Jolak, "Understanding software design for creating better design environments," 2017.

[9] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 492–501.

[10] D. Budgen, "The cobbler's children: Why do software design environments not support design practices?" in *Software Designers in Action: A Human-Centric Look at Design Work*. Chapman and Hall/CRC, 2013, pp. 199–216.

[11] S. Abrahão, R. F. Paige, S. Kokaly, B. Cheng, F. Bordeleau, H. Störrle, and J. Whittle, "User experience for model-driven engineering: Challenges and future directions," in *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, 2017.

[12] N. L. Chervany and D. Lending, "Case tools: understanding the reasons for non-use," *ACM SIGCPR Computer Personnel*, vol. 19, no. 2, pp. 13–26, 1998.

[13] L. Fowler, J. Armarego, and M. Allen, "Case tools: Constructivism and its application to learning and usability of software engineering tools," *Computer Science Education*, vol. 11, no. 3, pp. 261–272, 2001.

[14] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "A taxonomy of tool-related issues affecting the adoption of model-driven engineering," *Software & Systems Modeling*, vol. 16, no. 2, pp. 313–331, 2017.

[15] P. Mohagheghi and V. Dehlen, "Where is the proof?–a review of experiences from applying MDE in industry," *Lecture Notes in Computer Science*, vol. 5095, no. 2008, pp. 432–443, 2008.

[16] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014.

[17] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Assessing the state-of-practice of model-based engineering in the embedded systems domain," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2014, pp. 166–182.

[18] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio, "Relevance, benefits, and problems of software modelling and model driven techniques—a survey in the italian industry," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2110–2126, 2013.

[19] D. Kamma and S. K. G, "Effect of Model Based Software Development on Productivity of Enhancement Tasks – An Industrial Study," *2014 21st Asia-Pacific Software Engineering Conference*, pp. 71–77, 2014. [Online]. Available: http://ieeexplore.ieee.org/document/7091293/

[20] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[21] N. Mellegård, A. Ferwerda, K. Lind, R. Heldal, and M. R. V. Chaudron, "Impact of Introducing Domain-Specific Modelling in Software Maintenance: An Industrial Case Study," *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 248–263, 2016.

[22] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill *et al.*, "The relevance of model-driven engineering thirty years from now," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2014, pp. 183–200.

[23] P. Baker, S. Loh, and F. Weil, "Model-driven engineering in a large industrial context—motorola case study," *Model Driven Engineering Languages and Systems*, pp. 476–491, 2005.

[24] J. D. Herbsleb, "Global software engineering: The future of socio-technical coordination," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 188–198.

[25] D. Šmite and C. Wohlin, "A whisper of evidence in global software engineering," *IEEE software*, vol. 28, no. 4, pp. 15–18, 2011.

[26] A. A. Khan, J. Keung, M. Niazi, S. Hussain, and M. Shameem, "Gsepim: A roadmap for software process assessment and improvement in the domain of global software development," *Journal of software: Evolution and Process*, vol. 31, no. 1, p. e1988, 2019.

[27] R. Jolak, T. Ho-Quang, M. R.V. Chaudron, and R. R.H. Schiffelers, "Model-based software engineering: A multiple-case study on challenges and development efforts," in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 2018, pp. 213–223.

[28] D. L. Moody, "The" physics" of notations: a scientific approach to designing visual notations in software engineering," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2. IEEE, 2010, pp. 485–486.

[29] S. Meliá, C. Cachero, J. M. Hermida, and E. Aparicio, "Comparison of a textual versus a graphical notation for the maintainability of mde domain models: an empirical pilot study," *Software Quality Journal*, vol. 24, no. 3, pp. 709–735, 2016.

[30] B. Tversky, "Visualizing thought," *Topics in Cognitive Science*, vol. 3, no. 3, pp. 499–535, 2011.

[31] ——, "What do sketches say about thinking," in *2002 AAAI Spring Symposium, Sketch Understanding Workshop, Stanford University, AAAI Technical Report SS-02-08*, 2002, pp. 148–151.

[32] N. Mangano, T. D. LaToza, M. Petre, and A. Van Der Hoek, "How software designers interact with sketches at the whiteboard," *Software Engineering, IEEE Trans. on*, vol. 41, no. 2, pp. 135–156, 2015.

[33] M. Petre, A. van der Hoek, and Y. Quach, *Software Design Decoded: 66 Ways Experts Think*. MIT Press, 2016.

[34] I. Hammouda, H. Burden, R. Heldal, and M. R. V. Chaudron, "Case tools versus pencil and paper: A student's perspective on modeling software design." in *EduSymp@ MoDELS*, 2014, pp. 21–30.

[35] S. Baltes and S. Diehl, "Sketches and diagrams in practice," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 530–541.

[36] F. Tomassetti, M. Torchiano, A. Tiso, F. Ricca, and G. Reggio, "Maturity of software modelling and model driven engineering: A survey in the italian industry," 2012.

[37] B. Moggridge and B. Atkinson, *Designing interactions*. MIT press Cambridge, MA, 2007, vol. 17.

[38] P. Cohen and S. Oviatt, "Multimodal interfaces that process what comes naturally," *Commun ACM*, vol. 43, no. 3, pp. 45–33, 2000.

[39] S. Oviatt, P. Cohen, L. Wu, L. Duncan, B. Suhm, J. Bers, T. Holzman, T. Winograd, J. Landay, J. Larson *et al.*, "Designing the user interface for multimodal speech and pen-based gesture applications: state-of-the-art systems and future research directions," *Human-computer interaction*, vol. 15, no. 4, pp. 263–322, 2000.

[40] C. Ebert, M. Kuhrmann, and R. Prikladnicki, "Global software engineering: Evolution and trends," in *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*. IEEE, 2016, pp. 144–153.

[41] M. Franzago, D. Di Ruscio, I. Malavolta, and H. Muccini, "Collaborative model-driven software engineering: a classification framework and a research map," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1146–1175, 2017.

[42] A. von Mayrhauser and A. M. Vans, "Industrial experience with an integrated code comprehension model," *Software Engineering Journal*, vol. 10, no. 5, pp. 171–182, 1995.

[43] M.-A. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Systems and Software*, vol. 44, no. 3, pp. 171–185, 1999.

[44] A. J. Ko, H. H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks," in *the 27th International Conference on Software Engineering*. IEEE, 2005, pp. 126–135.

[45] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[46] P. Ralph, "Comparing two software design process theories," in *International conference on design science research in information systems*. Springer, 2010, pp. 139–153.

[47] D. Budgen, *Software design*. Pearson Education, 2003.

[48] K. Dorst and N. Cross, "Creativity in the design process: co-evolution of problem–solution," *Design studies*, vol. 22, no. 5, pp. 425–437, 2001.

[49] N. Cross, *Design thinking: Understanding how designers think and work*. Berg, 2011.

[50] T. Lindberg, C. Meinel, and R. Wagner, "Design thinking: A fruitful concept for IT development?" in *Design thinking*. Springer, 2011, pp. 3–18.

[51] F. P. Brooks Jr, *The design of design: Essays from a computer scientist*. Pearson Education, 2010.

[52] M. R. V. Chaudron, A. Fernandes-Saez, R. Hebig, T. Ho-Quang, and R. Jolak, "Diversity in UML modeling explained: Observations, classifications and theorizations," in *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 2018, pp. 47–66.

[53] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[54] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. ACM, 1974, pp. 249–264.

[55] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[56] B. Tversky, "Multiple models. in the mind and in the world," *Historical Social Research/Historische Sozialforschung. Supplement*, no. 31, pp. 59–65, 2018.

[57] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice," *Software & Systems Modeling*, vol. 17, no. 1, pp. 91–113, 2018.

[58] S. Kent, "Model driven engineering," in *Integrated formal methods*. Springer, 2002, pp. 286–298.

[59] M. Petre and A. Van Der Hoek, *Software Designers in Action: A Human-Centric Look at Design Work*. CRC Press, 2013.

[60] A. Tang, A. Aleti, J. Burge, and H. van Vliet, "What makes software design effective?" *Design Studies*, vol. 31, no. 6, pp. 614–640, 2010.

[61] B. Sharif, N. Dragan, A. Sutton, M. L. Collard, and J. I. Maletic, "Identifying and analyzing software design activities," in *Software Designers in Action: A Human-Centric Look at Design Work*. Chapman and Hall/CRC, 2013, pp. 153–174.

[62] A. Baker and A. van der Hoek, "Ideas, subjects, and cycles as lenses for understanding the software design process," *Design Studies*, vol. 31, no. 6, pp. 590–613, 2010.

[63] M. Razavian, A. Tang, R. Capilla, and P. Lago, "In two minds: how reflections influence software design thinking," *Journal of Software: Evolution and Process*, vol. 28, no. 6, pp. 394–426, 2016.

[64] K. Dorst, "The core of 'design thinking' and its application," *Design studies*, vol. 32, no. 6, pp. 521–532, 2011.

[65] D. Karis, D. Wildman, and A. Mané, "Improving remote collaboration with video conferencing and video portals," *Human–Computer Interaction*, vol. 31, no. 1, pp. 1–58, 2016.

[66] W. Heijstek, T. Kuhne, and M. R.V. Chaudron, "Experimental analysis of textual and graphical representations for software architecture design," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011, pp. 167–176.

[67] U. Dekel and J. D. Herbsleb, "Notation and representation in collaborative object-oriented design: an observational study," in *ACM SIGPLAN Notices*, vol. 42, no. 10. ACM, 2007, pp. 261–280.

[68] C. H. Damm, K. M. Hansen, and M. Thomsen, "Tool support for cooperative object-oriented design: gesture based modelling on an electronic whiteboard," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 2000, pp. 518–525.

[69] Q. Chen, J. Grundy, and J. Hosking, "An e-whiteboard application to support early design-stage sketching of UML diagrams," in *Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium on*. IEEE, 2003, pp. 219–226.

[70] J. Grundy and J. Hosking, "Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 282–291.

[71] N. Mangano, T. D. LaToza, M. Petre, and A. van der Hoek, "Supporting informal design with interactive whiteboards," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2014, pp. 331–340.

[72] D. Wüest, N. Seyff, and M. Glinz, "Flexisketch: A mobile sketching tool for software modeling," in *Mobile Computing, Applications, and Services*. Springer, 2012, pp. 225–244.

[73] W. Al Abed, V. Bonnet, M. Schöttle, E. Yildirim, O. Alam, and J. Kienzle, "Touchram: A multitouch-enabled tool for aspect-oriented software design," in *International Conference on Software Language Engineering*. Springer, 2012, pp. 275–285.

[74] F. Brieler and M. Minas, "Recognition and processing of hand-drawn diagrams using syntactic and semantic analysis," in *Proceedings of the working conference on Advanced visual interfaces*, 2008, pp. 181–188.

[75] S. Baltes, P. Schmitz, and S. Diehl, "Linking sketches and diagrams to source code artifacts," in *Proceedings of the the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 743–746.

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

[77] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," *Guide to advanced empirical software engineering*, pp. 285–311, 2008.

[78] R. K. Yin, *Case study research and applications: Design and methods*. Sage publications, 2017.

[79] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons, 2012.

[80] R. Wieringa, *Design Science Methodology for Information Systems and Software Engineering.* Springer, 2014.

[81] R. Kosara, C. G. Healey, V. Interrante, D. H. Laidlaw, and C. Ware, "Thoughts on user studies: Why, how, and when," *IEEE Computer Graphics and Applications*, vol. 23, no. 4, pp. 20–25, 2003.

[82] T. Gorschek, E. Tempero, and L. Angelis, "On the use of software design models in software development practice: An empirical investigation," *Journal of Systems and Software*, vol. 95, pp. 176–193, 2014.

[83] R. Jolak, B. Vesin, and M. R. V. Chaudron, "OctoUML: an environment for exploratory and collaborative software design," in *39th International Conference on Software Engineering. ICSE*, vol. 17, 2017.

[84] M. R. V. Chaudron and R. Jolak, "A vision on a new generation of software design environments," in *First Int. Workshop on Human Factors in Modeling (HuFaMo 2015). CEUR-WS*, 2015, pp. 11–16.

[85] R. Jolak, E. Umuhoza, T. Ho-Quang, M. R.V. Chaudron, and M. Brambilla, "Dissecting design effort and drawing effort in UML modeling," in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE, 2017, pp. 384–391.

[86] J. Sauro, *A practical guide to the system usability scale: Background, benchmarks & best practices.* Measuring Usability LLC, 2011.

[87] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernandez, B. Nordmoen, and M. Fritzsche, "Where does model-driven engineering help? experiences from three industrial cases," *Software & Systems Modeling*, vol. 12, no. 3, pp. 619–639, 2013.

[88] R. Jolak, K.-D. Le, K. B. Sener, and M. R. V. Chaudron, "Octobubbles: A multi-view interactive environment for concurrent visualization and synchronization of UML models and code," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2018, pp. 482–486.

[89] T. Hettel, M. Lawley, and K. Raymond, "Model synchronisation: Definitions for round-trip engineering," *Theory and Practice of Model Transformations*, pp. 31–45, 2008.

[90] D. Damian and D. Moitra, "Guest editors' introduction: Global software development: How far have we come?" *IEEE software*, vol. 23, no. 5, pp. 17–19, 2006.

[91] E. H. Trainer and D. F. Redmiles, "Bridging the gap between awareness and trust in globally distributed software teams," *Journal of Systems and Software*, vol. 144, pp. 328–341, 2018.

[92] S. Jarboe, "Procedures for enhancing group decision making," *Communication and group decision making*, pp. 345–383, 1996.

[93] F. Kortum, J. Klünder, and K. Schneider, "Don't underestimate the human factors! exploring team communication effects," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2017, pp. 457–469.

[94] R. E. De Vries, B. Van den Hooff, and J. A. De Ridder, "Explaining knowledge sharing: The role of team communication styles, job satisfaction, and performance beliefs," *Communication research*, vol. 33, no. 2, pp. 115–135, 2006.

[95] L. W. Anderson, D. R. Krathwohl, P. W. Airasian, K. A. Cruikshank, R. E. Mayer, P. R. Pintrich, J. Raths, and M. C. Wittrock, "A taxonomy for learning, teaching, and assessing: A revision of bloom's taxonomy of educational objectives, abridged edition," *White Plains, NY: Longman*, 2001.

[96] A. Soller, "Supporting social interaction in an intelligent collaborative learning system," 2001.

[97] R. Jolak, A. Wortmann, M. R.V. Chaudron, and B. Rumpe, "Does distance still matter? insights from revisiting collaborative distributed software design," *IEEE Software*, 2018.

[98] R. Jolak, M. Savary-Lelanc, M. Dalibor, A. Wortmann, R. Hebig, J. Vincur, I. Polasek, X. Le Pallec, S. Gérard, and M. R.V. Chaudron, "Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication," *Empirical Software Engineering*, pp. in–submission, 2020.

[99] A. Tang, F. Bex, C. Schriek, and J. M. E. van der Werf, "Improving software design reasoning–a reminder card approach," *Journal of Systems and Software*, vol. 144, pp. 22–40, 2018.

[100] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez *et al.*, "On-demand developer documentation," in *2017 IEEE International Confer-*

ence on Software Maintenance and Evolution (ICSME). IEEE, 2017, pp. 479–483.

[101] M. Petre, "UML in practice," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 722–731.

[102] B. Anda, K. Hansen, I. Gullesen, and H. K. Thorsen, "Experiences from introducing UML-based development in a large safety-critical project," *Empirical Software Engineering*, vol. 11, no. 4, pp. 555–581, 2006.

[103] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius, "Empirical evidence about the UML: a systematic literature review," *Software: Practice and Experience*, vol. 41, no. 4, pp. 363–392, 2011.

[104] A. Nugroho and M. R. V. Chaudron, "A survey into the rigor of UML use and its perceived impact on quality and productivity," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 90–99.

[105] W. Mischel, E. B. Ebbesen, and A. Raskoff Zeiss, "Cognitive and attentional mechanisms in delay of gratification." *Journal of personality and social psychology*, vol. 21, no. 2, p. 204, 1972.

[106] D. Stikkolorum, T. Ho-Quang, and M. R. V. Chaudron, "Revealing students' UML class diagram modelling strategies with webUML and logviz," in *SEAA, 2015 41st Euromicro*. IEEE, 2015, pp. 275–279.

[107] D. R. Stikkolorum, T. Ho-Quang, B. Karasneh, and M. R. V. Chaudron, "Uncovering students' common difficulties and strategies during a class diagram design process: an online experiment." in *EduSymp@ MoDELS*, 2015, pp. 29–42.

[108] M. Brambilla and P. Fraternali, "Large-scale model-driven engineering of web user interaction: The WebML and WebRatio experience," *Science of Computer Programming*, vol. 89, Part B, pp. 71 – 87, 2014, special issue on Success Stories in Model Driven Engineering.

[109] O. Diaz and F. M. Villoria, "Generating blogs out of product catalogues: An MDE approach," *Journal of Systems and Software*, vol. 83, no. 10, pp. 1970 – 1982, 2010.

[110] M. Brambilla, A. Mauri, and E. Umuhoza, "Extending the interaction flow modeling language (ifml) for model driven development of mobile applications front end," in *International Conference on Mobile Web and Information Systems*. Springer, 2014, pp. 176–191.

[111] D. Klyve and L. Stemkoski, "Graeco-Latin Squares and a Mistaken Conjecture of Euler," *The College Mathematics Journal*, vol. 37, no. 1, 2006.

[112] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013, ISBN 3-900051-07-0. [Online]. Available: www.R-project.org

[113] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[114] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[115] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[116] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[117] R. Jolak, B. Vesin, M. Isaksson, and M. R. V. Chaudron, "Towards a new generation of software design environments: Supporting the use of informal and formal notations with OctoUML," in *Second International Workshop on Human Factors in Modeling (HuFaMo 2016). CEUR-WS*, 2016, pp. 3–10.

[118] M. R. V. Chaudron, W. Heijstek, and A. Nugroho, "How effective is UML modeling?" *Software & Systems Modeling*, vol. 11, no. 4, pp. 571–580, 2012.

[119] S. Oviatt and P. Cohen, "Perceptual user interfaces: multimodal interfaces that process what comes naturally," *Communications of the ACM*, vol. 43, no. 3, pp. 45–53, 2000.

[120] M. Magin and S. Kopf, "A collaborative multi-touch UML design tool," *Technical reports*, vol. 13, 2013.

[121] S. Lahtinen and J. Peltonen, "Adding speech recognition support to UML tools," *Journal of Visual Languages & Computing*, vol. 16, no. 1, pp. 85–118, 2005.

[122] J. Brooke *et al.*, "Sus-a quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[123] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis Lectures on Software Engineering*, vol. 1, no. 1, pp. 1–182, 2012.

[124] P. Ralph, "The two paradigms of software development research," *Science of Computer Programming*, 2018.

[125] E. Kocaguneli, T. Menzies, and J. W. Keung, "On the value of ensemble effort estimation," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1403–1416, 2012.

[126] R. S. Pressman, *Software engineering: a practitioner's approach (5th ed.)*. McGraw-Hill, 2000.

[127] S. W. Ambler, "A manager's introduction to the rational unified process (RUP)," *http://www.ambysoft.com/downloads/managersIntroToRUP.pdf*, 2005.

[128] B. W. Boehm, "Industrial software metrics top 10 list," *IEEE software*, vol. 4, no. 5, pp. 84–85, 1987.

[129] B. W. Boehm, R. Madachy, B. Steece *et al.*, *Software cost estimation with Cocomo II with Cdrom*. Prentice Hall PTR, 2000.

[130] F. P. Brooks Jr, "The mythical man-month (anniversary ed.)," 1995.

[131] M. V. Zelkowitz, "Perspectives in software engineering," *ACM Comput. Surv.*, vol. 10, no. 2, pp. 197–216, Jun. 1978. [Online]. Available: http://doi.acm.org/10.1145/356725.356731

[132] I. Sommerville, *Software Engineering 8*. Pearson Education, 2007.

[133] Y. Yang, M. He, M. Li, Q. Wang, and B. Boehm, "Phase distribution of software development effort," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2008, pp. 61–69.

[134] E. Papatheocharous, S. Bibi, I. Stamelos, and A. S. Andreou, "An investigation of effort distribution among development phases: A four-stage progressive software cost estimation model," *Journal of Software: Evolution and Process*, vol. 29, no. 10, 2017.

[135] ISBSG, "International Software Benchmarking Standards Group. The benchmark release 10," 2008, [Online; accessed 25-April-2018].

[136] W. Heijstek and M. R. V. Chaudron, "Effort distribution in model-based development," in *2nd Workshop on Model Size Metrics*, 2007.

[137] R. Van Der Straeten, T. Mens, and S. Van Baelen, "Challenges in model-driven software engineering," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 35–47.

[138] A. Forward and T. C. Lethbridge, "Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals," in *Proceedings of the 2008 international workshop on Models in software engineering*. ACM, 2008, pp. 27–32.

[139] A. Kuhn, G. C. Murphy, and C. A. Thompson, "An exploratory study of forces and frictions affecting large-scale model-driven development," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 352–367.

[140] R. E. Herriott and W. A. Firestone, "Multisite qualitative policy research: Optimizing description and generalizability," *Educational researcher*, vol. 12, no. 2, pp. 14–19, 1983.

[141] J. Kramer and O. Hazzan, "The role of abstraction in software engineering," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 1017–1018.

[142] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[143] M. D. Plumlee and C. Ware, "Zooming versus multiple window interfaces: Cognitive costs of visual comparisons," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 13, no. 2, pp. 179–209, 2006.

[144] R. Jolak, B. Vesin, and M. R.V. Chaudron, "Using voice commands for UML modelling support on interactive whiteboards: Insights and Experiences," in *CibSE@ ICSE*, 2017, p. In Print.

[145] T. C. Lethbridge, V. Abdelzad, M. H. Orabi, A. H. Orabi, and O. Adesina, "Merging modeling and programming using Umple," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2016, pp. 187–197.

[146] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr, "Code bubbles: a working set-based interface for code understanding and maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 2503–2512.

[147] D. Röthlisberger, O. Nierstrasz, and S. Ducasse, "Autumn leaves: Curing the window plague in IDEs," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 237–246.

[148] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, "The impact of UML documentation on software maintenance: An experimental evaluation," *IEEE Trans. Softw. Eng*, vol. 32, no. 6, pp. 365–381, 2006.

[149] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of UML in software maintenance," *IEEE Transactions on software engineering*, vol. 34, no. 3, pp. 407–432, 2008.

[150] E. Haapalainen, S. Kim, J. F. Forlizzi, and A. K. Dey, "Psychophysiological measures for assessing cognitive load," in *the 12th ACM int. conference on Ubiquitous computing*. ACM, 2010, pp. 301–310.

[151] J. Klingner, "Fixation-aligned pupillary response averaging," in *Proceedings of the 2010 Symposium on Eye-Tracking Research & Applications*. ACM, 2010, pp. 275–282.

[152] J. H. Goldberg and X. P. Kotval, "Computer interface evaluation using eye movements: methods and constructs," *International Journal of Industrial Ergonomics*, vol. 24, no. 6, pp. 631–645, 1999.

[153] J. Scholtz, E. Morse, and M. P. Steves, "Evaluation metrics and methodologies for user-centered evaluation of intelligent systems," *Interacting with computers*, vol. 18, no. 6, pp. 1186–1214, 2006.

[154] G. M. Olson and J. S. Olson, "Distance matters," *Human–computer interaction*, vol. 15, no. 2-3, pp. 139–178, 2000.

[155] P. Bjørn, M. Esbensen, R. E. Jensen, and S. Matthiesen, "Does distance still matter? revisiting the cscw fundamentals on distributed collaboration," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 21, no. 5, p. 27, 2014.

[156] M. M. McManus and R. M. Aiken, "Monitoring computer-based collaborative problem solving," *Journal of Interactive Learning Research*, vol. 6, no. 4, p. 307, 1995.

[157] R. Weinreich, I. Groher, and C. Miesbauer, "An expert survey on kinds, influence factors and documentation of design decisions in practice," *Future Generation Computer Systems*, vol. 47, pp. 145–160, 2015.

[158] H. H. Clark, S. E. Brennan *et al.*, "Grounding in communication," *Perspectives on socially shared cognition*, vol. 13, no. 1991, pp. 127–149, 1991.

[159] P. S. Greenberg, R. H. Greenberg, and Y. L. Antonucci, "Creating and sustaining trust in virtual teams," *Business horizons*, vol. 50, no. 4, pp. 325–333, 2007.

[160] A. Monk, "Common ground in electronically mediated communication: Clark's theory of language use," *HCI models, theories, and frameworks: Toward a multidisciplinary science*, pp. 265–289, 2003.

[161] S. Kauffeld and N. Lehmann-Willenbrock, "Meetings matter: Effects of team meetings on team and organizational success," *Small Group Research*, vol. 43, no. 2, pp. 130–158, 2012.

[162] C. Gralha, D. Damian, A. I. T. Wasserman, M. Goulão, and J. Araújo, "The evolution of requirements practices in software startups," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 823–833.

[163] E. Bobek and B. Tversky, "Creating visual explanations improves learning," *Cognitive Research: Principles and Implications*, vol. 1, no. 1, p. 27, 2016.

[164] I. Rus, M. Lindvall, and S. Sinha, "Knowledge management in software engineering," *IEEE software*, vol. 19, no. 3, pp. 26–38, 2002.

[165] S. Cruz, F. Q. da Silva, and L. F. Capretz, "Forty years of research on personality in software engineering: A mapping study," *Computers in Human Behavior*, vol. 46, pp. 94–113, 2015.

[166] Z. Sharafi, A. Marchetto, A. Susi, G. Antoniol, and Y.-G. Gueheneuc, "An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 33–42.

[167] O. Liskin, "How artifacts support and impede requirements communication," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2015, pp. 132–147.

[168] M. E. Meade, J. D. Wammes, and M. A. Fernandes, "Drawing as an encoding tool: Memorial benefits in younger and older adults," *Experimental aging research*, vol. 44, no. 5, pp. 369–396, 2018.

[169] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 201–228.

[170] V. R. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 456–473, 1999.

[171] M.-A. Storey, A. Zagalsky, F. Figueira Filho, L. Singer, and D. M. German, "How social and communication channels shape and challenge a participatory culture in software development," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 185–204, 2017.

[172] B. Dobing and J. Parsons, "How UML is used," *Communications of the ACM*, vol. 49, no. 5, pp. 109–113, 2006.

[173] M. Borenstein, L. V. Hedges, J. P. Higgins, and H. R. Rothstein, *Introduction to meta-analysis*. John Wiley & Sons, 2011.

[174] T. K. Koo and M. Y. Li, "A guideline of selecting and reporting intraclass correlation coefficients for reliability research," *Journal of chiropractic medicine*, vol. 15, no. 2, pp. 155–163, 2016.

[175] L. V. Hedges, "Distribution theory for glass's estimator of effect size and related estimators," *journal of Educational Statistics*, vol. 6, no. 2, pp. 107–128, 1981.

[176] P. D. Ellis, *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge University Press, 2010.

[177] D. Falessi, L. C. Briand, G. Cantone, R. Capilla, and P. Kruchten, "The value of design rationale information," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, p. 21, 2013.

[178] A. Tang, M. A. Babar, I. Gorton, and J. Han, "A survey of architecture design rationale," *Journal of systems and software*, vol. 79, no. 12, pp. 1792–1804, 2006.

[179] J. Cohen, *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[180] M. Völter, "Md*/dsl best practices update march 2011," *Update*, 2011.

[181] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144–161, 2014.

[182] J. J. Ray, "The Construct Validity of Balanced Likert Scales," *The Journal of Social Psychology*, vol. 118, no. 1, pp. 141–142, 1982.

[183] M. B. Donnellan, F. L. Oswald, B. M. Baird, and R. E. Lucas, "The mini-ipip scales: tiny-yet-effective measures of the big five factors of personality." *Psychological assessment*, vol. 18, no. 2, p. 192, 2006.

[184] S. Conversy, "Unifying textual and visual: A theoretical account of the visual perception of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 2014, pp. 201–212.

[185] M. Höst, B. Regnell, and C. Wohlin, "Using students as subjects—a comparative study of students and professionals in lead-time impact assessment," *Empirical Software Engineering*, vol. 5, no. 3, pp. 201–214, 2000.

[186] F. I. Craik, "Aging and memory: Attentional resources and cognitive control," 2019.

[187] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo, "Empirical software engineering experts on the use of students and professionals in experiments," *Empirical Software Engineering*, vol. 23, no. 1, pp. 452–489, 2018.

[188] A. Wortmann, O. Barais, B. Combemale, and M. Wimmer, "Modeling languages in industry 4.0: an extended systematic mapping study," *Software and Systems Modeling*, Sep 2019.

[189] A. Casamayor, D. Godoy, and M. Campo, "Mining textual requirements to assist architectural software design: a state of the art review," *Artificial Intelligence Review*, vol. 38, no. 3, pp. 173–191, Oct 2012.

[190] S. Wagner and D. M. Fernández, "Analyzing text in software projects," in *The Art and Science of Analyzing Software Data*. Elsevier, 2015, pp. 39–72.

[191] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[192] T. Cant, J. McCarthy, and R. Stanley, "Tools for requirements management: a comparison of telelogic doors and the hive," Defence Science and Technology Organisation Edinburg (Australia) Information Networks DIV, Tech. Rep., 2006.

[193] M. Wolf, M. Petridis, and J. Ma, "Using structural similarity for effective retrieval of knowledge from class diagrams," in *International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Springer, 2013, pp. 185–198.

[194] A. Shatnawi, A. Seriai, H. Sahraoui, and Z. Al-Shara, "Mining software components from object-oriented apis," in *International Conference on Software Reuse*. Springer, 2015, pp. 330–347.

[195] S. J. Guastello, "Creative problem solving groups at the edge of chaos," *The Journal of Creative Behavior*, vol. 32, no. 1, pp. 38–57, 1998.

[196] A. L. Ramos, J. V. Ferreira, and J. Barceló, "Model-based systems engineering: An emerging approach for modern systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 1, pp. 101–111, 2012.

[197] K. Hölldobler, B. Rumpe, and A. Wortmann, "Software Language Engineering in the Large: Towards Composing and Deriving Languages," *Computer Languages, Systems & Structures*, vol. 54, pp. 386–405, 2018.

[198] B. Rumpe, *Agile Modeling with UML: Code Generation, Testing, Refactoring.* Springer International, May 2017.

[199] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Commun. ACM*, vol. 31, no. 11, pp. 1268–1287, Nov. 1988.

[200] M. Pikkarainen, J. Haikara, O. Salo, P. Abrahamsson, and J. Still, "The impact of agile practices on communication in software development," *Empirical Software Engineering*, vol. 13, no. 3, pp. 303–337, Jun 2008.

[201] D. Karlström and P. Runeson, "Integrating agile software development into stage-gate managed product development," *Empirical Software Engineering*, vol. 11, no. 2, pp. 203–225, Jun 2006.

[202] J. Wu, T. C. N. Graham, and P. W. Smith, "A study of collaboration in software design," in *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.*, Sep. 2003, pp. 304–313.

[203] D. Graziotin, X. Wang, and P. Abrahamsson, "Do feelings matter? on the correlation of affects and the self-assessed productivity in software engineering," *Journal of Software: Evolution and Process*, vol. 27, no. 7, pp. 467–487, 2015.

[204] A. Tang, M. H. Tran, J. Han, and H. Van Vliet, "Design reasoning improves software design quality," in *International Conference on the Quality of Software Architectures.* Springer, 2008, pp. 28–42.

[205] J. Singer and N. G. Vinson, "Ethical issues in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1171–1180, 2002.