Designing a Projectional Editor for Live Coding

Designing a Projectional Editor for Live Coding

Using Design Thinking to Improve Teaching

By

Maryam Hosseinkord, M.Sc.

A Thesis

Submitted to the Department of Computing and Software

and School of Graduate Studies

of McMaster University

in the Partial Fulfillment of the Requirements

for the Degree Master of Applied Science

McMaster University © Copyright by Maryam Hosseinkord

December 16, 2021

All Rights Reserved

McMaster University

Master of Applied Science (2021)

Hamilton, Ontario (Department of Computing and Software)

TITLE: Designing a Projectional Editor for Live Coding

Using Design Thinking to Improve Teaching

AUTHOR: Maryam Hosseinkord M.Sc.(Artificial Intelligence), Research and Science

University of Tehran

SUPERVISOR: Dr. Christopher Anand

NUMBER OF PAGES: xiii, 89

Abstract

How can observation of a legacy system be used for design? To answer this question, we observed a teacher doing live coding with a conventional code editor and used the observations to design an editor better suited to this style of teaching. In particular, we found strong evidence that a projectional editor would better meet this need. Reflecting on this experience, we describe two types of requirements which can be inferred from observing a user using a legacy system: hidden requirements, in which users use existing features in unexpected ways, and novel requirements inferred from pain points observed in current system use.

Acknowledgements

I would like to extend my sincere thanks to my supervisor Dr. Christopher Anand, who guided and assisted me during my studies at McMaster University. Thanks to my family, who gave me love and stood by my side during this period of my life. Thanks to all my friends, specially Nasim, Azin, and Atifeh, for taking care of me.

Contents

Abstract iii

Acknowledgements iv

Acronyms x

Declaration of Authorship xiii

1 Introduction 1

1.1 Purpose 2

1.2 Research Questions (RQ) 3

1.3 Contributions 3

1.4 Structure of Thesis 3

2 Literature Review 5

2.1 A Brief Look at the History of Design Thinking 6

2.2 d.school Design Thinking Model 8

2.3 Design Thinking and Software Development 11

2.3.1 Why does software fail? 13

2.3.2 Understanding User's Hidden Needs 14

2.3.3 Understanding Users' Requirements 15

2.3.4 Other Areas That Design Thinking (DT) Can Help 17

2.4 Parser-Based Editors vs Projectional Editors 21

2.4.1 From Syntax-Controlled Systems to Projectional Editors 22
2.5 Projectional Editors and Teaching 25
2.5.1 Design Principles and Challenges 26

3 Methods **31**
3.1 Empathy 31
3.1.1 Ethnography 31
3.1.2 Video Ethnography 33
3.1.3 Ethnography Challenges 35
3.1.4 Recording Data (What-How-Why) 36
3.1.5 Personas 37
3.2 Define 38
3.2.1 Sensemaking 38
3.2.2 Point of Views 39
3.2.3 How Might We Questions 40
3.3 Ideate 40
3.3.1 Brainstorming 41
3.3.2 Brainstorm Selection 42

4 Results **43**
4.1 Observations and Note Taking 43
4.2 Recording the Data 44
4.3 Personas 47
4.4 Defining the problem 50
4.5 Ideation and Brainstorming 51

5 Discussion And Design **60**
5.1 Projectional Editor or Not? 61
5.2 General Editing Requirements 65

5.2.1 Selecting . . . 67
5.2.2 Deleting . . . 68
5.2.3 Copying and Cutting . . . 69
5.2.4 Pasting . . . 69
5.2.5 Undo and Redo . . . 70
5.2.6 Find and Replace . . . 70
5.2.7 Commenting . . . 71
5.2.8 Out of Order Editing Habits . . . 71
5.3 The User Interface . . . 73
5.4 Teaching Functionalities . . . 76

6 Conclusion **81**
6.1 Research Questions . . . 81
6.2 Next Steps . . . 82

Bibliography **84**

List of Figures

2.1 d.school's The Design Thinking Model . . . 9

4.1 Student Persona . . . 48

4.2 Instructor Persona . . . 49

4.3 Grouping Observations . . . 51

5.1 Suggested User Interface . . . 73

5.2 Suggested Menu . . . 80

List of Tables

2.1 Comparing different Design Thinking Models. 8
2.2 Editor Challenges introduced by Völter. 28
2.3 Hansen's Principles of Design applied in EMILY 29
2.4 Norman's Principles of Design 30

4.1 Results 53
4.2 Results-continue 54
4.3 Results-continue 55
4.4 Results-continue 56
4.5 Results-continue 57
4.6 Results-continue 58
4.7 Results-continue 59

Acronyms

ACM Association for Computing Machinery

AST Abstract Syntax Tree

DBLP DataBase systems and Logic Programming

DT Design Thinking

ERP Enterprise Resource Planning

FRS Functional Requirements Specifications

GNOME Gandalf NOvice prograMming Environment

HMW How Might We

IDEO Innovation Design Engineering Organization

IEEE the Institute of Electrical and Electronics Engineers

JavaRDISE Java Rookies Driven Into a Structured Editor

MPS Meta Programming System

POV Point of View

RE Requirement Elicitation

RQ Research Questions

TA Teaching Assistant

UCD User Centered Design

UI User Interface

Declaration of Authorship

I, Maryam Hosseinkord, declare that this thesis titled, ”Designing a Projectional Editor for Live Coding Using Design Thinking to Improve Teaching” and the work presented in it are my own.

Chapter 1

Introduction

In recent years, several initiatives have been created to encourage people of all ages to learn how to code, and many countries have revised their computing curricula to include coding skills. However, teaching and learning coding are both very challenging tasks. Research has identified numerous difficulties that students face when learning to code, for example, the need for a wide range of skills, including problem-solving, understanding correct syntax, and debugging [52]. Additionally, factors like students' distractions, lack of a suitable curricula to engage students, and lack of motivation and effort from students make teaching coding a challenge. Along with all these identified determinants, we cannot ignore the role of the code editors in students' and teachers' classroom experience. Therefore, a need for better tools for coding is felt more than ever.

User Centered Design (UCD) seems a promising approach for creating solutions with high usability and hopefully great user acceptance. UCD is an iterative design process in which designers emphasize the users' needs as they progress through each phase. Design Thinking (DT) as a UCD method has gained popularity in designing software solutions recently. In an attempt to create a more user-friendly code editor, we conducted an ethnography-based study adopting DT methods to design a coding tool that can assist

both teachers and learners.

1.1 Purpose

This study focuses on designing a code editor appropriate for teaching the Elm programming language to first-year-university Computer Science students. While there is extensive literature on designing different code editors, few papers focus specifically on designing a code editor for teaching or learning purposes, and none focus on live coding.

To find requirements and design a good solution, we decided to use DT as a user-centered approach. Iterative DT begins with empathy for users, diverges into different definitions for the problem, converges on a focused problem, then diverges into possible solutions and converges on the right solution. The user then tests a rapid prototype and provides feedback. This process continues until enough user satisfaction is achieved.

In terms of adopting DT techniques in different software engineering projects, a large amount of literature exists. However, most research focuses on how DT combined with Agile methodologies can be applied in software development projects. We have reviewed some examples of how DT has improved software development processes in Chapter 2. However, none of the reviewed papers explicitly outlined the steps they took before their design decisions. Furthermore, the research rarely mentions a legacy system or how it affected the design process, although it would seem natural that a legacy system is taken into account in the first iteration of DT. Thus, this study aims to describe the experience of designing a code editor using DT, including all steps in detail and reviewing the effects of the legacy system. The output of this thesis is the design for a prototype which needs to be implemented and used to gather feedback and better understand our users, starting the process of DT iterations.

1.2 Research Questions (RQ)

- **RQ1** Can observation of a legacy system be used to design a novel code editor specifically used in live coding by an instructor?
- **RQ2** What features should an editor designed for teaching have?
- **RQ3** Is there evidence supporting the usefulness of a Projectional Editor by the instructor in teaching coding?
- **RQ4** What kind of requirements can we get from observing a legacy system?

1.3 Contributions

The main contributions of this study include:

1. Documenting a case study observing a legacy system to create an initial prototype as part of a design cycle, explaining the strengths and weaknesses of this approach.
2. Imagining a code editor specifically designed for teaching via live coding.
3. Discussing our design decisions with respect to Norman's design principles
4. Creating personas whose points of view should be used in analysis and feature design.

1.4 Structure of Thesis

The rest of the thesis is structured as follows: In Chapter 2, we review the relevant literature on DT and Software Development using DT, discussing some of the challenges that can be tackled using this methodology and providing some examples. Chapter 3 describes the research methodologies employed in this study, explaining that different

methods can be utilized to complete each phase of DT. Chapter 4 presents the results of this study, revealing how the data was gathered, classified, and analyzed. This chapter reviews the results of the first three steps of the DT process: empathize, define, and ideate. Chapter 5 demonstrates our proposed design for the first prototype and discusses our design decisions. Finally, Chapter 6 summarizes the thesis and suggests directions for future research.

Chapter 2

Literature Review

The word "design" has three different meanings: a field of design, a conceptual proposal, and an actual product. In addition, it can also be used as a verb to describe an activity. Here are the three most relevant definitions.

First, we use the term "design" to describe the outcome of a design project, i.e., the changes brought about by it. A change can be many things:

- An improvement to an existing solution, like an airbag or a computer mouse roller.
- A change in a product's physical appearance.
- A change in how the item performs.
- The act of creating something completely new, whether it be a physical object, a service, or a system, is also a change.

Second, this change is usually conceptualized before it is implemented. The design is a theory or conceptual idea about what may be valuable to people. Finally, a design refers to a process or activity intended to instigate these changes [55].

2.1 A Brief Look at the History of Design Thinking

In 1969, the Nobel Prize-winning computer scientist Herbert A. Simon mentioned design as a science or way of thinking in his book, "Sciences of the Artificial," [45] for the first time. His book argues convincingly for setting up courses in design, concluding with a blueprint for a curriculum in design alongside natural science in the whole engineering curriculum [2]. Simon presented one of the first formal models of the Design Thinking (DT) process in his book. His model had seven major phases, each consisting of component stages and activities. His remarks mention rapid prototyping and testing through observation, which forms the core of today's many designs and entrepreneurial processes [12].

His model consisted of 7 different stages.

- Define: Decide what issue you are trying to resolve. Agree on who the audience is. Prioritize this project in terms of urgency. Determine what will make this project successful and establish a glossary of terms.
- Research: Review the history of the issue. Note existing obstacles. Collect examples of other attempts to solve the issue. Note the project supporters, investors, and critics. Talk to your end users for fruitful ideas for later design and consider thought leaders' opinions.
- Ideate: Identify the needs and motivations of your end users. Generate as many ideas as possible to serve identified needs. Record your brainstorming sessions. Do not judge or debate ideas during brainstorming; have one conversation at a time.
- Prototype: Combine, expand, and refine ideas. Create multiple drafts. Get feedback from a diverse group. Include end users. Present a selection of ideas to the

client. Reserve judgement and maintain neutrality. Create and present actual working prototype(s).

- Choose: Review the objective. Set aside emotion and ownership of ideas. Avoid consensus thinking.
- Implement: Make task descriptions. Plan tasks. Determine resources. Assign tasks. Execute and deliver to client.
- Learn: Gather feedback from the consumer. Determine if the solution met its goals. Discuss what could be improved. Measure success. Collect data and document.

In 1987, Harvard's Director of Urban Design Programs, Peter Rowe, published his book, "Design Thinking," [54] and discussed how an architect approaches a design task in detail. He provided case studies on three different designers resulting from long periods of observation and documentation.

Innovation Design Engineering Organization (IDEO) was founded by Stanford University professor David Kelley and introduced its design process in 1991. As one of the pioneers of DT, IDEO has developed its terminology, processes, and toolkits over the years. IDEO is obviously not the inventor of DT, but they have become known for solving problems on different scales. IDEO provides a 6-step iterative process for applying Human-Centered Design. These steps are Observation, Ideation, Rapid Prototyping, User Feedback, Iteration, and Implementation [25].

Another design process mentioned in their Human-Centered Design Field Guide has three steps. "Inspiration, Ideation, and Implementation." This model was proposed by Tim Brown [42], [36].

The Hasso Plattner Institute of Design at Stanford, commonly known as the d.school, was founded by Stanford professor David M. Kelley and his colleagues in 2004. Since its

<table>
<tr><th>Model</th><th colspan="7">Steps</th></tr>
<tr><td>Simon</td><td>Define</td><td>Research</td><td>Ideate</td><td>Prototyping</td><td>Choose</td><td>Input</td><td>Learn</td></tr>
<tr><td>IDEO</td><td colspan="2">Observation</td><td>Ideate</td><td>Rapid Prototyping</td><td colspan="3">Implementation</td></tr>
<tr><td>Brown</td><td colspan="2">Inspiration</td><td>Ideate</td><td colspan="4">Implementation</td></tr>
<tr><td>d.school</td><td>Empathy</td><td>Define</td><td>Ideate</td><td>Prototyping</td><td colspan="3">Test</td></tr>
</table>

TABLE 2.1: Comparing different Design Thinking Models.

inception, they have made the development, teaching, and implementation of DT their own central goals. They have proposed a five-stage DT process consisting of "Empathy, Define, Ideate, Prototype, and Test" [6].

As reviewed above, many varieties of DT are used today, and while they may have a different number of stages, ranging from three to seven, they are all based on the same principles as Simon's model. Table 2.1 shows a comparison of mentioned DT models. Note that even though "Define" used to take place before "Research" in the Preliminary Model by Simon, the next models show no clear border between Research and Define, and finally, d.school's model clearly states that the Empathy/Research step should happen before defining the problem to ensure that it is the correct problem. In this study, we focus on the five-stage DT model proposed by d.school.

2.2 d.school Design Thinking Model

DT is an exploratory approach to problem-solving that includes and balances both analytical and creative thought processes [31]. As explained in section 2.1 DT has been modeled in several different ways. The model that we use in this study is Stanford's d.school Design Thinking Model, as shown in Figure 2.1.

Phase 1: Empathize

Empathy is the first step in DT. By empathizing with users, the researcher should understand the problem and share their feelings within the context of the problem. By the end

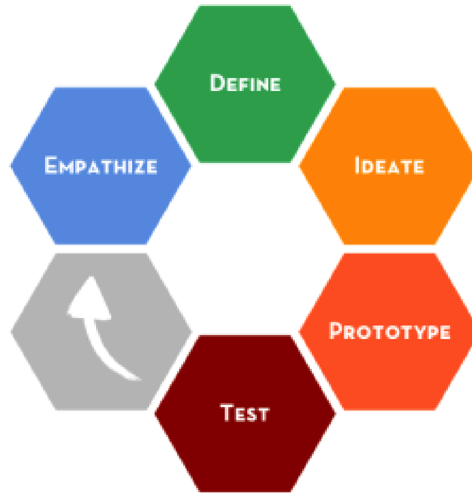

Figure 2.1: The Design Thinking Model, image from Stanford's d.school.

of this step, the researcher should be able to answer some important questions or report specific information about the user's feeling, the problem space, and the environment. These questions pertain to the user's emotions and physical needs, their thoughts, their statements and their actions, as well as the intention behind and meaning of each action. In order to gain such wisdom and to collect such information, the researcher should closely observe users and take careful notes. There are different methods of observing the user, such as observing them in real-life environments or video recording them [28], [38].

Phase 2: Define

For the design space to be clearer and more focused, you must use precise, accurate, meaningful terms to describe the problem. While defining the problem, the researcher should consider the results from the previous steps, such as the user and context of the problem. At this point, the researcher should synthesize the gathered information and gained empathy in order to formulate a meaningful and actionable problem statement.

To put it simply, the Define mode is about sense making [28].

Phase 3: Ideate

As soon as you have completed the problem statement and know what problem needs to be solved, you can begin to brainstorm. Ideation is when you concentrate on generating a wide range of ideas. In this step, the researcher should brainstorm different solutions to the problem and create as many as possible. As the ideas developed in this step fuel prototyping, designers must unleash their creativity and begin imagining solutions. Sometimes even impossible ideas, which one may consider irrational, may provoke a perfectly actionable solution. There is no single correct approach to finding a great idea. However, brainstorming, mind mapping, and sketch notes are a few strategies to consider [49].

Phase 4: Prototype

The prototype phase involves creating artifacts to solve the design challenge. Design decisions for the prototype are derived from the ideas found during the ideation phase. Although generating as many ideas as possible is recommended during the ideation phase, we only carry a few of them forward to the prototyping phase since we must create an artifact for each idea. Therefore, it is advised that you select only two to three ideas, make design decisions based on them, and create artifacts. A prototype allows designers to gain a deeper understanding of the aspects of the proposed solution and gain insight for the following iterations.

Considering that DT is an iterative process, the prototypes should be easy to make, not very expensive, and not time-consuming. The term prototype can refer to anything that allows users to interact with it—be it a wall of sticky notes, a gadget you have created, a role-playing exercise, or even a storyboard [28]. It is important to analyze all the prototyping experiences to learn from our mistakes.

Phase 5: Test

Testing is the process of asking your potential users to test your prototype. During this mode, the researcher should carefully observe each user's behavior and collect their feedback. A valuable form of testing is when the user works with the prototype in a real environment without the researcher's involvement. Having gained empathy from step 1, the designer now understands the user's behavior better, and, optimistically, their point of view is closer to the user's. Testing is not about whether users like your solution or not; it is about what lessons can be learned from their feedback to improve your solution in the following iterations. Prototyping allows you to avoid spending money and time on solutions that are not optimal.

It is important to prototype as if you are certain you are correct and test as if you are sure you are wrong. Testing is the opportunity to refine your solution and make it better [28].

2.3 Design Thinking and Software Development

Extensive literature exists when it comes to adopting DT techniques in different software engineering projects. However, most of the research focuses on the combination of DT and Agile methodology in software development projects.

Some systematic reviews on the subject identify the effectiveness of the combination of DT with Agile methodology. For example, a study in 2018 [41] presented a literature review to evaluate how DT is integrated with Agile methods. They selected 29 studies to answer their research questions. Their results show that integrating DT and Agile has resulted in more customer satisfaction with the products. The study also reports improvements in usability, supporting the proper management of challenges and requirements discovery.

Some researchers focused on understanding why software companies have applied DT in their projects and what techniques they have found more useful. A study [8] did an empirical experiment by asking 59 Agile teams from Brazilian software organizations about adapting DT techniques in Agile projects. Their research showed that DT techniques were mostly used for Requirement Elicitation (RE) purposes, and the most adopted techniques by the participants were prototyping, brainstorming, and interviews.

Another study [46] surveyed 127 professionals from the Brazilian software industry, asking about their experience applying different DT models in different scenarios. Their study revealed that professionals follow more than ten different models (sets of steps). Among more than 50 techniques used by the participants, brainstorming, personas, and empathy maps were the most commonly used. Based on this study, 94.49% of the participants chose “generating ideas and solutions” as a big motivator to use DT, followed by “explore and understand the problem” and “to create innovative idea” as the second and third place.

Hugo Martins et al. conducted a [17] systematic literature review in 2019 to answer their research question about the challenges of eliciting requirements and methods using DT. Their research consisted of a keyword search in the Digital Library Association for Computing Machinery (ACM), the Institute of Electrical and Electronics Engineers (IEEE) Xplore, and DataBase systems and Logic Programming (DBLP) Computer Science Bibliography, and a manual search through conferences, newspapers, and magazines. They answered their research questions based on the top 20 selected studies they listed. In addition, they listed 31 DT methods that were mentioned to be useful in the selected papers.

2.3.1 Why does software fail?

According to Harvard Business School professor Clayton Christensen, over 30,000 new products are introduced every year, and 95 percent fail. Therefore, new products often fail. Even though many software companies use Agile frameworks, many software projects still fail [34]. They fail both in the manufacturing and service sectors. Our definition of failure is when these new products fail to excite customers and do not reach sales objectives or market-share targets set by the companies that develop them. The way top companies "listen" to their customers is changing as managers realize that end users are often unable to articulate their hidden needs [19]. We will discuss hidden needs in section 2.3.2.

Studies show that new products and services fail primarily due to being too similar to existing offerings. Products that are difficult to differentiate do not attract the attention of customers. However, the lack of distinguishing features is just a symptom of the real problem. The real problem is the lack of understanding of customers' needs. A fundamentally different approach is needed if companies are to be successful at identifying real customer needs [19].

Mirza et al. [34] suggest that the main reason behind the production of wasteful software is that the software teams fail to understand the user requirements completely. Focusing on this idea, they did a study to answer the research question: "How can we reduce developing wasteful software?" They hypothesized that gaining empathy with the user can solve this problem by helping the developer understand the user requirements more deeply, resulting in meaningful products. They used a combined Agile framework and DT approach to create an application to memorize things more easily. Even though they did not report any empirical evidence on how effective their method was, they claimed that this approach could create good customer satisfaction and help in reducing the production of wasteful applications that the customer does not use.

De and Vijayakumaran [11] believe most products fail due to an incomplete understanding of their prospective end users. The lack of interaction between the development team and the end users makes it easy to overlook this lack of understanding, resulting in products that do not meet end-user needs. The framing of prototyping as a process of learning about the users is DT's attempt to inoculate designers from this danger. They also look at the same problem from a software tester's perspective and focus on how DT methods can provide a way to reduce and identify the bugs early. For example, the use of personas (discussed in depth below) in creating test cases is a concrete way of ensuring that the development process considers different users' needs at all stages of development.

In summary, we have found three reasons mentioned in related literature for product failures:

1. Failing to understand the user's hidden needs.

2. Failing to understand the user's requirements.

3. Failing to identify the end user.

Many DT and software studies focus on customers' hidden (item 1) and known requirements (item 2). In the next section, we review some of them.

2.3.2 Understanding User's Hidden Needs

Goffin [19] defines hidden needs as "issues and problems that customers face but have not yet realized." Identifying hidden needs is a big challenge but a big opportunity to add value. Addressing hidden needs in a product design will highly increase customer satisfaction. It has been pointed out that hidden needs are those which "many customers recognize as important in the final product but do not or are not able to articulate in

advance." [19]. One of the promising areas that software engineering can benefit from DT tools and techniques is "finding hidden needs."

Even though the term "hidden" was not used directly in reviewed papers, some research finds evidence of DT methods being used to uncover hidden features in their software projects.

In one study [38], the team reports that some features were added to the system while not explicitly asked for but derived from observations and references made during the empathy phase.

In addition, they state that the team uncovered some features after creating the first prototype and going through continuous cycles of development and feedback with participants. They were asked several questions regarding the suggestions and comments made by other participants, and this process allowed the team to receive multiple feedback points about potential new features and changes that were not perceived in the previous iterations.

2.3.3 Understanding Users' Requirements

The success of a software system is primarily determined by its ability to fulfill the purpose for which it was developed. Requirements engineering (RE) is the process of discovering that purpose by identifying stakeholders and their needs, and then documenting them in a form that can be analyzed, communicated, and implemented [40].

The importance of understanding end-user needs and incorporating them into the software development process is widely recognized in software engineering. In modern software development, the user is included in many ways, but even though user stories should represent a user's needs, they frequently communicate the product owner's or software development team's viewpoints [33]. IBM research by Lucena et al. in 2016

[33] proposes an approach focused on satisfying end-user needs employing DT iterative software development. Their framework is divided into two main phases. The Visioning Phase is responsible for developing software requirements using several DT practices, and The Delivery Wave consists of software development Sprints. They applied their methodology in five real software development projects at IBM. Their experiment revealed that the up-front analysis and user feedback resulting from the DT framework offers a better understanding of what problems need to be solved and the best solutions to satisfy the user needs—as a result of that 80% of the surveyed reported end users had a very high satisfaction rate on the projects delivered.

A look into User Requirements Classifications:

User's needs are classified by [19] into three categories:

- **Known** needs are customer needs that have been identified for some time. Available products already address this group of needs.
- **Unmet** needs are needs recognized by customers but not currently satisfied by existing products.
- **Hidden** needs are needs that users have not yet discovered.

Also Requirements can be classified based on their explainability [3]:

- **Explicit**: These requirements are clearly expressed, and the system must meet them.
- **Tacit**: These requirements are not directly expressed or captured but are essential to meet System's goal. They are (1) hard to express, convert, communicate and share; (2) often related to application domain; (3) described as users' tacit knowledge; (4) are experiential knowledge which developing team accumulates step by

step in practice during a long period; (5) are hard to encode and articulate; (6) can be expressed hazily and crudely". These requirements are met in the system but cannot be explicitly described.

In this study, we preferred to use the first group terminology since it classifies the requirements from a user's knowledge perspective and matches better with the user-centered nature of our methodology. We do not use the concept of unmet requirements in this study because unmet and hidden requirements are observed almost the same during observations. Unmet requirements are different from hidden ones only when you interview users and ask them about their needs. Our case study only included observation of the legacy system, and since both hidden and unmet requirements are missing from the system, we can approach them the same.

We decided not to use the concept of tacit and explicit requirements because they describe the requirements considering legacy system functionalities. Both explicit and tacit requirements are met in the legacy system, but tacit ones are hard to extract or describe. We needed terminology that could differ the available and non-available features in the legacy system. Therefore, we found this concept different from hidden requirements, which are unknown to the user, missing from the legacy system, and might only be discovered by identifying new pain points via observation.

2.3.4 Other Areas That DT Can Help

Some papers focus on pointing out how DT techniques can improve the software development process in general. In the following pages, we will review some of them.

The Problem Space Definition

One of the challenges of understanding a wicked problem is that no single observer can claim to have fully analyzed and understood the full scope of the problem [5]. Thus,

one of the areas where DT can extensively help software engineering is problem space definition.

While applying the DT method on their case study, Newman et al. [38] stated that creating a substantial number of physical and digital technology prototypes was very useful in understanding the problem space; they believed that building each prototype helped designers explore a different aspect of the problem space. Borrowing an idea from Burnett and Myers [5], they argued that presenting a technological solution to a community was not good enough; instead, the designers should explore the problem space together with the user group.

"It is not good enough to simply present the community with a technological solution to a complex problem; instead, the problem space needs to be jointly explored with the community."

The physical artifacts created in the prototyping stage in DT are used as boundary objects that help the designers focus on the problem space and provide a technical scope.

On the other hand, Jensen et al. [27] stated that applying DT in the early stages of product development at SAP was highly effective because DT allowed for rapid iterative development and resulted in a faster reframing of the actual problem. Furthermore, SAP projects benefit from DT to gain a holistic overview of a problem. Hence every project always starts with a discovery phase researching customers, end users, and current technological solutions.

Unbalanced Knowledge Distribution

Heikkila et al. published a study on requirements engineering in Agile software development [24]. They state that "unbalanced knowledge distribution" is a challenge in Agile

software development. Because Agile practices rely on highly skilled people, most knowledge remains tacit [24]. DT can help in this regard. Since DT supports a team-based approach to requirements gathering, knowledge is more evenly distributed throughout the team. According to them, involving interdisciplinary teams in the DT process, the whole group understands the users' needs. Furthermore, various perspectives enable a more comprehensive elicitation that reveals the tacit knowledge of stakeholders and team members [23].

Feature Prioritization

The study mentioned above [24] also identifies the difficulties in prioritizing requirements as a challenge in Agile software development. Even though they did not find DT to be helpful in this area, another study [38] reported that in their case study, the group discussions among participants in the DT process were particularly useful for system feature prioritization since the participants could directly encounter each other and talk about how a feature is important or not.

Neglecting Non-functional Requirements

Husaria and Guerreiro [26] review the practices of RE and suggest how DT could have a role in mitigating the challenges that RE could have. For example, they report that "neglecting non-functional requirements" is a challenge in Agile software development, and DT can help with it. According to them, DT draws considerable attention to non-functional requirements by emphasizing the importance of usability requirements.

Accessing the User

Customer or user issues are a challenge of Agile RE. Accessing and interacting directly with customers is difficult, which delays the process of clarifying requirements. As DT is

a process-oriented method, user interviews can be planned in advance to help overcome this problem [24].

On the other hand, contrary to the traditional RE that tends to limit the involvement of users in the development process by rigorous documentation, Agile software development tends to have a significant back and forth relationship with stakeholders. DT can provide many tools for such purposes [26].

User Interface Design:

Many publications report the effectiveness of DT in user interface design in different software practices. We review the most relevant to our study.

A study by Muraer from the BMW group [35] describes the application of DT for the user interface design for smart glasses. They involved BMW workers as real end users and invented a photo prototyping method inspired by paper prototyping. They report that using this method generated many different solutions in a short period that surpassed the known solutions in the existing literature.

Another study [7] applies DT in redesigning two legacy systems for the Brazilian Military by adding the prototyping phase and facilitating a designer's participation throughout the development life cycle. Their results reflected the importance of the role of the designer and the quality of proposed prototypes. In particular, 67% of survey participants replied that they fully agree that it is the designer's role to offer an easy-to-memorize interface. Furthermore, 96% of survey participants agree on the positive experience of using prototypes to support the development of project features.

Suzianti et al. focused on using the DT method to develop a mobile application for tsunami disaster management in Indonesia [48]. In addition to empathizing with the user, they interviewed one of the tsunami victims. Even though they couldn't continue

developing their mobile application to the final result, their research predicts a high chance of acceptance of the application within the community.

Suzianti and Arrafah applied DT to redesign an Enterprise Resource Planning (ERP) system for a Dental Clinic [47]. The redesign was successful, and the new User Interface (UI) addresses the needs of each user, enabling him to do his work more efficiently. Validation of the final product was done by clinical personnel as the end users. Additionally, researchers justified the prototype for its lower time on tasks while preventing some errors and increased user satisfaction compared to the current UI by including some survey instruments as measurement instruments during the DT process.

When using DT, researchers found that software engineers' viewpoints should be considered during user interface redesign [47]. Therefore, they recommend that they be included as an additional persona to ensure the feasibility of the redesigned system. Furthermore, they pointed out that DT solutions still must follow certain rules to be applicable. Hence, a design system or guidelines (such as Google Material Design and Design System) need to be used.

2.4 Parser-Based Editors vs Projectional Editors

In parser-based code editors, the users type the code in a text buffer, and then a parser parses the code to check if it conforms to a grammar. The parser creates an Abstract Syntax Tree (AST), which contains the program's structure. However, a projectional editor does not require a parser. Whenever a user edits a program, the AST is directly modified. As the user interacts with the AST, the projection engine creates a representation that reflects the changes resulting from the interaction. Developers often find projectional editing to be challenging. Although projectional editors have existed for a long time, they have not been widely adopted in practice. It may be due to the problems associated with the editor's usability and design [53]. We believe that the literature

that explains the design of user-friendly parser-based code editors may contribute to the design of better projectional editors since the study of code editors has a long history.

In this section, we provide a brief description of how the related terminology evolved, and then we summarize two important studies that provide design guidelines for code editors and projectional editors that inspired us.

2.4.1 From Syntax-Controlled Systems to Projectional Editors

Attempts to help developers to eliminate syntax errors date back to 1971. William J. Hansen developed the Emily Interactive Syntax-Controlled System for creating and manipulating program texts. Emily used the syntax of the programming language to impose a tree structure on programs in the language [21].

Hansen claims [20]: "Many systems for construction and modification of computer programs exist, but all existing systems require the programmer to enter his text as a sequence of characters. With Emily, the user constructs his text by selecting choices from the menu to replace certain symbols in the text."

In Emily, programs begin as single non-terminal symbols. Then the system presents all of the syntax rules that define replacements for each non-terminal. These replacement strings generally contain characters and non-terminals, and the users create programs by selecting appropriate rules. Because the editor preserves the tree structure of the text, it is possible to manipulate it according to its structural units.

It is clear from this explanation that the concept he was working on was very similar to projectional editors, but he did not decide to assign it a specific name. He worked on Emily until 1984 and published multiple papers and manuals about it.

It was at the end of 1974 that Donzeau-Gouge and her colleagues [14] began working on the same concept. They aimed to bridge the gap between existing programming tools

and the vast amount of theoretical research on the semantics of programming languages. Their system was implemented using Pascal. One of the areas of their investigation was close to the concept of the projectional editors. Their approach was to debug the code by source-level program manipulation. Donzeau-Gouge implemented her system named MENTOR, a processor designed to manipulate ASTs. MENTOR is driven by a tree manipulation language MENTOL.

Donzeau-Gouge used the phrases "Compiler Assisted Programming" and "Structure Oriented Editing" in her publication in 1975 [13] , and she called MENTOR a "Syntax-directed Editor" in 1984 [15]. Later in 1985, Gouge introduced MENTOR-Ada, an instance of MENTOR which inherits all the qualities of MENTOR. In MENTOR-Ada, manipulations were based on representing programs as trees, similar to other structure-oriented systems.

In 1986, Bernard Lang studied MENTOR's potential in the context of teaching, software development and maintenance, and language design. According to him, the use of a syntax-directed environment for trivial tasks is costly. However, it is not clear if the benefits of using this technology are sufficient to outweigh the costs involved, both in terms of the complexity of the processors and computing power required, and the additional difficulty for users in learning more complex tools. There are some benefits for novice users since syntax-directed environments may be viewed as a helpful learning tool since it is structured at a beginner level and may help novice users learn new languages [32].

The first steps in preventing syntax errors involved designing a source code editor that understood the programming language grammar and used this knowledge to improve editing and program execution. This ensures that a program is always syntactically correct. In the early years, there was no strict rule for whether to include a parser or not, but they generally discussed editing the language tree rather than the text.

Therefore, even though the term "Syntax Directed Editor" is used in some publications, they refer to a tightly coupled editor and incremental parser.

In May 1978, Teitelbaum started designing and implementing the well-known "Cornell program synthesizer." Synthesizer's editor is a combination of a text editor and a tree editor. In this editor, the user generates templates by commands, and they can type expressions and assignment statements one character at a time. Templates are predefined, so errors are not possible. The editor detects errors instantly because it invokes the parser phrase-by-phrase as the user types. They claim that the synthesizer has syntax-directed program editing features of EMILY and tree editing features of MENTOR and other existing systems [50].

Although "syntax-directed editing" was the first term to describe projectional editing, it never gained popularity and was rarely used in publications after 1993.

Eric Sandwell coined the term "structured editor" in his paper. As part of his paper, he reviewed existing programming methodologies in the Lisp user environment, emphasizing methods for interactive program development. Based on the observed behavior of Lisp users, he believed that top-down programming could be done more effectively not only by using stepwise refinement but often through what has been called structured growth, which was a disciplined approach to changing the programs [53].

Later, the term "projectional editor" gained traction following its use by Fowler in 2008. Finally, Voelter's use of this term in the paper "Embedded software development with projectional language workbenches" introduced this term to the academic literature, although "structure editor" is still widely used [4].

2.5 Projectional Editors and Teaching

When it comes to projectional editors and teaching, we could find only a few studies. There is evidence that structured editors have been used in several universities for teaching since the late 1970s. However, education has never widely adopted this promising concept.

In 1977, MENTOR was used in some universities to teach Pascal. As Lang reported in [32], younger learners and young engineers from companies collaborating with their research group adapted very easily to syntax-directed editing regardless of their prior training. In contrast, some older professionals struggled with the syntax-directed approach.

Garlen and Miller introduced Gandalf NOvice prograMming Environment (GNOME) in 1984 [18]. Carnegie-Mellon University was using a family of structure editors designed by their team to teach programming to undergrads at that time. In their study, the researchers used their structured editors' experience in a practical novice programming environment design. They discussed the lessons learned from adapting structure editors and the effectiveness of GNOME as a teaching environment. The authors state that GNOME makes returning to a programming language easier with the syntax leads and can aid developers in returning to a programming language without having to re-read the manuals. Moreover, 46 high school teachers participating in the study found GNOME enabled them to focus more on computer science concepts rather than the details of "getting it to work."

Among the most important factors in Garland and Miller's success with GNOME was their editor generator tool. The editor generator made it easy to create several editors and incrementally add more and more features to a basic editor. It also included the ability to integrate external tools into their environment.

The Java Rookies Driven Into a Structured Editor (JavaRDISE) projectional editor was introduced by Santos in 2020 [43]. His main intention behind designing this projectional editor was to remove the syntax barrier faced by beginning programmers. Based on his design, JavaRDISE covered the syntax requirements of the first programming course taught at his institution, Instituto Universitário de Lisboa (ISCTE–IUL). His paper discussed how JavaRDISE could benefit a classroom. He believes exposing novices to more syntax than necessary to teach fundamental concepts is counterproductive, and once they acquire a solid understanding of semantics, learning syntax gets easier for them. Santos believes that the biggest challenge for JavaRDISE, and structured editors in general, is the lack of usability. It is impossible to achieve the goal of easing introductory programming without providing a pleasant and productive way of manipulating the source code. Unfortunately, he did not conduct any controlled experiments, and the editor was not used in classrooms.

2.5.1 Design Principles and Challenges

In [53] Völter et al. conducted a study to understand why projectional editors have not seen much adoption in practice. Their hypothesis is that the main reason for this is the incompatibility of editor usability with infrastructure integration. Among the drawbacks are the unfamiliar editing experience and integration challenges. In their paper they investigate the usability of projectional editors. They used JetBrains Meta Programming System (MPS) as a case study and evaluated the effectiveness of MPS solutions for these issues by surveying professionals. Their study shows that flexible language composition and diverse notations can result in serious usability problems, which can be mitigated by using facilities that emulate the editing experience of parser-based editors.

The researchers identified 14 usability issues related to entering code efficiently, selecting and modifying code, and integrating with existing infrastructure. The authors

claim half of these issues are easily resolved using code completions or expression tree refactorings. Many others require language- or notation-specific implementations or are not mitigated conceptually. According to the survey, developers perceive projectional editing as an efficient technique for everyday work, though it requires a considerable amount of effort to learn. However, the survey also reveals weaknesses, such as weak support for commenting, which is not adequately addressed in MPS. Table 2.2 is reformatted based on the summary table provided in [53].

In [20] Hansen outlines a set of user engineering principles that were applied in the development of the Emily text editing system. He emphasized "Know the User" as the first user engineering principle, implying that Emily's design decisions have been guided by User Centered Design (UCD) principles. Table 2.3 shows his ideas.

Norman's Principles

Donald Norman, considered one of the all-time great researchers in human-computer interaction and UCD, provides six key design principles to keep in mind as you design any interface. Table 2.4 shows his ideas [39].

	Issue	Mitigation Technique used by MPS
Efficient Entering (Textual) Code	Requires manual, user-based disambiguation	code completion, aliases, context constraints
	Cannot establish references to non-existing nodes	intentions to create missing targets
	Requires structure-aware typing	side transforms, delete actions, smart references, wrappers, smart delimiters
	What you see is not what you type	
	Requires notation-specific editor support	(but editors share common aspects)
Selecting and Modifying Code	Selection is based on the tree structure	-
	Hard to perform cross-tree modification	expression tree restructuring
	Requires structure-aware copy/paste	paste handlers
	Does not support free-floating comments	(partly addressed by metamodel extension in mbeddr)
	Requires dedicated support for commenting code	(partly addressed by metamodel extension in mbeddr)
	Does not support custom layout	-
Infrastructure Integration	Requires tool support for diff/merge	node-by-node revert, merge driver, diff/merge tool using projection rules
	Text-based shell-scripting tools cannot be used	(build system support for generating and testing models)
	Requires tool support to export/import textual syntax	copy/paste, parser hooks, generic node(de-)serialization

TABLE 2.2: Editor Challenges introduced by Völter.

Principle	n	Decision	Explanation
Minimize Memorization	1	Selection not entry	If the choices displayed cover the user's needs, he can enter information more quickly by selection.
	2	Names not numbers	When the user is to select from a set of items, he should be able to select among them by name
	3	Predictable behavior	The system ought to have a 'personality' around which the user can organize their perception of the system.
	4	Access to system information	Any system is controlled by various parameters and keeps various statistics. The user should be given access to these and should be able to modify from the console any parameter that he can modify in any other way.
Optimize Operations	5	Rapid execution of common operations	Operations should be fast or at least give the user feedback. For instance, while printing a file Emily displays the line number of each tenth line as it is printed.
	6	Display inertia	This means the display should change as little as necessary to carry out a request.
	7	Muscle memory	One implication of muscle memory is that the meaning of specific interactions should have a simple relation to the state of the system. A button should not have more than a few state dependent meanings and one button should be reserved to always return the system to some basic control state.
	8	Reorganize command parameters	Observation of users in action will show that some commands are not as convenient as their frequency warrants while other commands are seldom used. Inconvenient commands can be simplified while infrequent commands can be relegated to sub-commands.
Engineer for Error	9	Good error messages	Error messages should be specific, indicating the type of error and the exact location of the error in the text.
	10	Engineer out the common errors	If an error occurs frequently, it is not the fault of the user, it is a problem in the system design.
	11	Reversible actions	A single erroneous deletion can inadvertently remove a very large substructure from the file. To protect the user the system must provide reversible actions.
	12	Redundancy	This simply means that the system provides more than one means to any given end.
	13	Data structure integrity	A system should provide sufficient Data Structure Integrity that regardless of system or hardware troublesome version of the user information will always be available.

TABLE 2.3: Hansen's Principles of Design applied in EMILY

#	Principle	Description	Example
1	Visibility	Users should be aware of their options and how to use them by just looking at an interface. Additionally, it is essential to ensure that only the options necessary are included.	In a sign-in screen, the information needed is only about logging in and signing up, so adding too much information would interfere with the visibility principle.
2	Feedback	After every action, the user should receive feedback indicating whether the action was successful.	A spinning icon on the tab indicates that a webpage is loading.
3	Affordance	Affordance is the relationship between how things look and how they can be used. A good interface should be intuitive enough so that users can access the desired information just by looking at it.	Coffee mugs have high affordance because you immediately know how to hold them.
4	Mapping	In a good design, the controls should closely resemble their effects.	The vertical scroll bar indicates where you are currently, and the page moves down at the same speed and sensitivity.
5	Constraints	Interface constraints limit certain forms of user interaction. This is essential because a wide range of options within an interface may overwhelm the user.	A phone number field on an online form that does not allow users to type letters.
6	Consis-tency	When people recognize patterns, they learn new things more easily. Users need consistency to recognize and learn these patterns. When similar-looking things produce different results, the user will become frustrated.	If a backward arrow designates the back button, it cannot be changed because it would be inconsistent with what the user has learned.

Table 2.4: Norman's Principles of Design

Chapter 3

Methods

This chapter describes the different methods we adopted for each step of the Design Thinking (DT) process.

3.1 Empathy

The problems a design thinker tries to solve are rarely their own. Instead, they belong to a particular group of people, the end users. As mentioned previously, gaining empathy with these people, and understanding their needs and feelings, is an important component of DT. The best solutions come out of the deepest insights into human behaviour.

Learning to recognize those insights is not an easy process because human minds automatically filter out much information. We need to learn to see things “with a fresh set of eyes,” rather than just looking, and empathizing is what can help [28].

3.1.1 Ethnography

The most significant point behind the empathy phase in User Centered Design (UCD) methods is the effort to understand the product or service concerning its contexts and

evaluate the design based on its impact on users' lives. The Collins COBIULD dictionary (1987) defines "context" in the following way:

"The context of something consists of the ideas, situation, events, or information that relate to it and make it possible to understand it fully. If something is seen in context or if it is put into context, it is considered with all the factors that are related to it rather than just being considered on its own, so that it can be properly understood." [55]

This definition makes clear that designers need to gain wisdom on a product and all the different ways it relates to its environment in everyday life. Careful observation and perception of context help the designer to come up with new ideas to feed the next steps of the process [55].

At the very first stage of a DT process, the designer should be able to collect all this information from the interviews and observations. Many methods already exist to help him in this stage. Ethnography originally developed and popularized in social sciences is one of the main qualitative research methods, that can help in this area [1].

The concept of ethnography (literally, "description of people") describes a study conducted by direct observation of users in their natural environment rather than in a lab. This type of research aims to learn how users interact with their natural environments. Defined by Van Dijk in his famous paper "Design ethnography: Taking inspiration from everyday life" [51]:

> Design ethnography is qualitative ethnographic research set within a design context. Design ethnography is aimed at understanding the future users of a design, such as a certain service. It is a structured process for going into the depth of the everyday lives and experiences of the end-users. The intention is to enable the design team to identify with these people, to build up an

empathic understanding of their practices and routines and what they care about.

Design ethnography is about original, rather than simulated, understanding of the users' practice [55].

A big advantage of ethnography over interviews is the honest and pure picture that ethnography can provide. In an interview, the users describe their work or leisure. So, naturally, they may say things they want the design team to hear. Due to this phenomenon, the ideas become filtered by the users' verbal expression skills and their expectations regarding the design team's intentions [55].

3.1.2 Video Ethnography

Today with a drastic increase in the complexity level of demanded products and services, it is difficult for a designer to understand the context of a problem. This situation gets even worse for customized services or specialized products. A designer needs to understand users' actions, intentions, values, and needs. They need to learn how users interact with a system and how the solution fits into practice in the real environment. The growth of numerous elements that the ethnographer should observe and perceive raises the need for better tools to support the ethnography process.

The use of videos can help ethnographers achieve their goals. The video ethnography process involves recording a stream of activity of subjects in their natural environment to investigate, interpret, and represent culture and society [44].

Video reveals behavior that would otherwise remain hidden due to its ability to observe for a long period. Researchers can leave the camera recording while they leave the scene and conduct research unobtrusively. Using this method can be useful in situations

where participants' work could be disrupted. Video provides access to some scenes that would otherwise be impossible to analyze in depth [55].

The use of video can be advantageous to ethnographic research in multiple ways [44]:

- With videos, a wide range of activities can be observed in their natural settings over an extended period. This coverage can complement written accounts and provide the context for the limited coverage of other methods.
- Video analysis allows for scientific rigor when performed by trained researchers. In addition, videos preserve the sequence of observed behavior for later examination, enhancing the validity and reliability of statements about the activity.
- Videos can be viewed by researchers and participants, allowing the scope of interpretation to grow.
- Videos can be used to connect abstractions and inferences to the observable activities that they are based on.

According to Heath and Luff, video has three characteristics that make it ideal for studying the interactional structure of workplace activities [22]:

- Video enables access to the nuances of the conversation and visual behavior, allowing for extensive investigation of the activities in slow motion if necessary.
- Researchers can exchange results with others via video recordings, allowing for discussion of the materials used in the investigation.
- Video allows for the public display of the findings. Therefore, it can be observed and examined by the public too.

3.1.3 Ethnography Challenges

Just like any other research method, there are challenges to ethnographic research. It's important to consider these before choosing the right research method. Here are some points to keep in mind:

1. Ethnography is very time-consuming and takes a lot of effort from the researcher to go to the new environment and learn about it. However, video ethnography eases this challenge by eliminating the need for the physical presence of the researcher in the environment.

2. Results from ethnographic research only apply to the people being studied. Therefore, it is not easy to generalize those results to other societies or situations. In our case, we should consider if our findings only apply to online classes or first-year students, or to similar courses or teaching styles.

3. Ethnography is subject to interpretation. A researcher's interpretation of the data may be biased.

4. There are practical and ethical concerns of ethnographic research. However, getting the prior consent of the participants, maintaining their confidentiality, and a proper research design can mitigate these issues.

5. The presence of a video camera affects people's behavior; therefore, the researcher might not capture an honest picture. Ethnographers face this challenge regularly. Using hidden cameras is one solution, but ethical concerns usually require that the ethnographer get consent from the participants before filming. Therefore, it is almost impossible to omit the video's effect on people's behavior.

3.1.4 Recording Data (What-How-Why)

What-How-Why is a method for helping researchers explore the observations and derive deeper understandings of people as they observe them. Using this method, we start with observation—the What—then move to abstraction—asking How—until finally, we arrive at the Why, which is the emotional underpinnings of the observed behavior. This method is recognized to be extremely useful for analyzing images [6], and we found it useful when analyzing our video recordings

There are straightforward instructions to use this method provided by the d.school [6]. We have followed the same steps to gather and organize our data. We set up an Excel sheet with three columns: What? How? and Why?

What: Starting with concrete observations

The instructor recorded the entire lecture for the use of students in the class, including both live coding and background discussions. We did our best to be objective and avoid making any assumptions by taking note of all the facts as they were.

How: Move to understand

We took notes on how the instructor was doing each specific action. Noting the required effort, whether they were rushed or had difficulty completing the action. Did the activity or situation appear to be impacting the user's state of being either positively or negatively? We used descriptive phrases packed with adjectives. This section allowed us to write about the emotional impact on the user based on the clues we saw.

Why: Time for interpretation

This section is where we tried to note "why the user is doing the specific action and why in that particular way?" This step usually requires that the researchers make informed guesses regarding motivation and emotions. They will discover assumptions

to test with users later and will often discover unexpected realities about a particular situation.

3.1.5 Personas

When designing for a broad audience of users, one temptation is to make the product's functionality as broad as possible to accommodate as many people as possible. But this approach is flawed. Instead, the correct approach is to design for specific types of users with particular needs [10]. Personas can help us identify types of users.

In 1999, Cooper introduced the concept of personas in his book "Inmates are running the asylum [9]." Cooper described a persona as "a collection of realistic representative information that may include fictitious details for a more accurate characterization." In simpler words, personas are based on reality, but some imaginary elements with no direct impact on the real product design are added to make the persona more tangible.

In summary, personas help us focus on the right individuals to design for, whose needs represent a larger group of key constituents. They also help us prioritize design elements to address the needs of the most important users without significantly hindering secondary users [10].

There is a complete process for creating personas in [10] that we followed:

1. Identify persona data sources.
2. Set up user categories: A user category is a group of users that share similar characteristics. In this step, we must find out related user categories for our product.
3. Collect user data: After deciding the categories of users, we can collect user data from data sources. In this step, we create skeletons.

4. Prioritize the skeletons: It's impossible to create personas for every skeleton. Therefore, we prioritize them according to their importance to our product.

5. Convert skeletons to persona foundation documents: Skeletons represent basic information about a user. But a persona foundation document 4.1 is a concrete narration of a specific user in written form. A variety of templates are available online. We used the template provided by mural.com for our research.

6. Designate persona types: Although there are multiple types of personas, our study only focuses on primary and secondary types. We need to decide which persona is the primary one.

3.2 Define

The define stage of the design process focuses on bringing clarity and focus to the design space. At this stage, designers should identify the challenge they intend to conquer based on what they have learned about the user and the context. After understanding the context and gaining invaluable empathy with the user through the previous step, now the designer can make sense of the vast amount of information gathered [28].

The define mode aims to present a problem statement that is both meaningful and actionable—this is called a point of view. A point of view is a guiding statement that focuses on the needs of a specific user and defines the right challenge to address.

3.2.1 Sensemaking

Sensemaking has multiple interesting definitions. In their paper [29], Klein et al. have gathered many of them while introducing their own definition. "Sensemaking is a motivated, continuous effort to understand connections (which can be among people, places, and events) to anticipate their trajectories and act effectively."

Sensemaking refers to people continually and relatively automatically going through an act-oriented process to integrate experiences into their understanding of their world. Designers make explicit this automatic process during the design synthesis phase as they interpret and model data to make sense of it [30].

Even though sensemaking does not follow a rigid formula, several methods can facilitate this process [6]. Regardless of the technique, here, the design team develops a shared understanding of the data gathered. In this step, the designers create a series of artificial constraints that informs but does not completely determine the design space under study. Using these artificial constraints, designers can frame a problem within a flexible framework. The design team can reframe the situation with sensemaking and define the correct design problem.

We used some of the methods suggested by Sandford's d.school to conduct our research.

3.2.2 Point of Views

Writing Point of View (POV) is a way of reframing a design challenge into an actionable problem statement. A good POV provides focus, frames the problem, and informs criteria for evaluating competing ideas. Additionally, it will allow the researcher to think more directly by creating How Might We (HMW) questions based on the POVs [28], [6].

The bootleg guide provided by Hasso Plattner offers multiple methods for expressing POVs [6]. In our research, we have utilized the POV Madlib method. Using the following Madlib, we captured and harmonized three elements of a POV: user, need, and insight.

[USER] needs to [USER'S NEED] because [SURPRISING INSIGHT]

Using this method, the designer tries out various options, trying out different combinations of variables. The insight and need should arise from your unpacking and synthesis work in empathy. "Needs" should be verbs, and insights should not simply be reasons for the need but rather a statement that can be used in the design process [6].

3.2.3 How Might We Questions

A HMW question is a short question intended to initiate brainstorming. After having the POV statement, a HMW can be considered as a seed for the ideation phase. Designers need to develop a seed that is broad enough to allow for a wide selection of solutions yet narrow enough to inspire the team to come up with specific, unique solutions. Note that the scope of the seed will differ depending on the project [6].

HMW questions are created from a point of view. The objective is to generate actionable questions that retain the unique and specific perspective of the POV. These questions can begin with "How might we..." and explore various enhancement opportunities within the problem space [6]. Additionally, Marty Neumeier's book [37] has suggested a few other question types to inspire the POVs. According to him, "In What Ways Might We...", "What's stopping us from...?" and "What would happen if...?" can be used to continue the brainstorming process. To generate different HMWs, this study used d.school's bootleg guide [6] as well as Neumeier's suggestions [37].

3.3 Ideate

Ideation is the mode of the design process that focuses on generating ideas. Ideation helps designers transition from identifying problems to developing solutions for the users. The ideation mode allows the designers to combine their understanding of the problem space and users with their imagination to generate solutions. Especially early in a design project, ideation is about seeking the widest range of possible solutions rather

than simply settling on one. Testing and feedback from users will determine the best solution later.

Many techniques are available to generate ideas, such as bodystorming, mind mapping, and sketching. In all of them, however, there is a common theme: deferring judgment—that is, separating idea generation from idea evaluation. Thus, the designer reinforces their imagination and creativity, knowing that they will get to the evaluation later. [28]

3.3.1 Brainstorming

During a brainstorming session, team members can generate many ideas that they would not have thought of if they were working alone. The purpose of brainstorming is to utilize the group's collective thinking by interacting with one another, listening to each other, and building upon previous ideas. As part of an effective brainstorm, the team should also intentionally turn up the generative part of their brains and turn down the evaluative part. In other words, we are seeking as many creative solutions as possible without judging them. Team members are encouraged to speak up about their ideas, even if they seem costly, time-consuming, or unfeasible.

The design team leader should be deliberate about setting aside a period for the team to operate in "brainstorm mode." This time does not need to be very long. If the team is highly engaged, 15 to 30 minutes should be sufficient. Team members can conduct this activity on a whiteboard or around a table while standing or sitting upright in an active posture. Using a HMW question is a great way to frame a brainstorm. All the ideas generated by the team are collected in this process [6].

3.3.2 Brainstorm Selection

Brainstorming should generate many diverse ideas. At the selection stage, we assess these ideas, which is called harvesting. It is fairly straightforward to gather ideas from brainstorming sessions and select a few, but when designing solutions, we should be careful about how we select the ideas. It is possible to carry forward a range of these ideas to preserve the breadth of solutions and to not settle for only the most convenient solution. We do not narrow our choices too quickly during the selection process. A design team can use different selection techniques, but in this study, we selected ideas by discussion within a group of two. If there was more than one interesting solution, we forwarded multiple ideas into prototyping [6].

Chapter 4

Results

4.1 Observations and Note Taking

To conduct this research, we analyzed videos recorded during the lectures of a first-year undergraduate course (1XD3 - Introduction to Software Design Using Web Programming) in the computer science program at McMaster University in Winter 2021 held online due to Covid-19 pandemic. Each lecture was approximately 45 minutes long and was conducted twice a week. During these sessions, the instructor taught concepts along with coding while he shared his code editor screen with the students. Additionally, the instructor's face could be seen in a small window at the bottom right corner of the screen. The settings for all the videos were identical. Students were able to communicate with the instructor through Microsoft Teams' chat box or audio chat. For this study the 7 first sessions were selected to be observed.

During the Covid-19 pandemic, all lectures, labs, and office hours were conducted via Microsoft Teams. Therefore, all the videos used in this study were filmed in neutral settings and the users' natural environments. Additionally, we believe that the video recording effect on users was minimal, as these videos were intended to be used for future use by students, and not as a reference to our observations.

Both the students and the instructor were made aware that the sessions was being recorded and anyone in the MS Team could view the recording. Only the instructor's image and voice were recorded in the live coding videos used in this study. Students answered questions via text chat, but the chat data was not used in note-taking. Since the purpose of the study is the improvement of course delivery, by TCP2 2018, Article 2.5 [16] it is exempt from ethics review. Note also that the instructor is a collaborator on this improvement project.

4.2 Recording the Data

To record the data, we used the table format of the what-how-why method and were interested in any actions and statements in the online class.

When it comes to recording data, it might seem straightforward. We might expect the researcher to sit and take note of everything. There are two considerable points associated with this idea. First, is it possible to take note of everything? Is it even good to end up with a huge list of observations that might be irrelevant? What are the criteria to distinguish relevant and non-relevant observations? The answer to these questions can help the researcher to avoid a burst of information. Second, what is everything? How do you define everything? The human mind filters much information it receives from the environment. How can we claim that the researcher's brain did not filter any valuable information? In sum, the ideal results come from recording as many relevant ideas as possible. It is desirable to keep irrelevant ideas to the minimum while we don't miss relevant and meaningful points.

Facing the challenge, our first session of observation was the most time-consuming one. It took four hours to watch a 45-minute video. Since we were unaware of what we might see in the video, what should be noted, and what is relevant, we took notes

of everything that caught our attention. In that state, we were highly focused on editing behaviors, and to avoid missing any valuable fact, we recorded every editing move performed by the instructor even if they seemed irrelevant at that moment.

Being inexperienced resulted in recording too many irrelevant notes. Therefore, many of the editing facts we recorded in early sessions did not end up contributing anything meaningful to our study. It is easy to say we should record all observations in the empathy step and leave the judgments to the future, but recording many irrelevant ideas is very disruptive for the researcher and will confuse them in the next steps. Therefore, it is good that the researcher gets more familiar with the problem in each observation and can take better notes over time. Having this in mind, they should do their best not to get into the observer's bias trap. Observer bias is a social-science term to describe the error introduced into measurement when observers overemphasize behavior they expect to find and fail to notice behavior they do not expect. Therefore, the observer should be aware that their mind may trick them into going forward to the solution-finding step and thereby emphasizing some observations over others.

Although the notes taken from the first observations should not contain many irrelevant facts, we believe the researcher should take as many notes as possible in the early stages of the study because they have no idea what is important and what is not.

For example, in one session, the instructor used the `List.map` function. This function takes a list, and a function, and applies the function to each item in the list and returns a list. As we observed, the instructor used an undefined function as input when using List.map. He later added the definition in the code. While this action may seem irrelevant to our design at first, we developed an important point of view resulting from it.

Additionally, our experience showed that we became more cautious about what factors to consider in subsequent sessions. Therefore we started to record more environmental facts compared to previous sessions. The sensory gating concept probably explains this phenomenon. Sensory gating or filtering is filtering out redundant or irrelevant stimuli from all the environmental stimuli that reach the brain. This process occurs automatically in the brain during attention processing when the brain selectively seeks information relevant to the goal. We believe that after learning about the different areas to focus on, our sensory gating began to disappear, and we started observing more environmental facts.

For example, we recorded that the instructor asks the students if the code is readable to them in the fourth session. We did not record this fact in previous sessions, but the instructor asked the same question at the beginning of every class. We believe that this fact was filtered out by our brain in previous sessions, while it was a very important point and finally led to a valuable, meaningful Point of View (POV).

Our experience has shown that multiple observation sessions are very effective since the observer becomes more familiar with the subject and can apply that knowledge to future observations.

The points that we recorded appeared with different frequencies. For example, using text editing functions such as copy, paste, and find, saving the code, running the code, and checking the output repeatedly occurred throughout each session as expected. Using the find function happened at least once in each of the sessions. Some observations occurred as the classes became more complex and more sophisticated structures were used. Some other observations occurred rarely or only once as a result of a coincidence or user error. The more videos we watched, the number of new observations to record reduced.

Prioritizing the requirements can be achieved by assessing the frequency of use of a feature or considering how frequently the user encounters a problem. The requirements we gathered are nearly all high-priority and should be included in the first prototype. We recorded 96 observations from 5 different sessions. We will discuss the classification of this data and sense-making in section 4.4.

4.3 Personas

As discussed in the Chapter 3, creating good personas is one of the desired outputs of the empathy phase. We created two role-based personas for this study and included effective factors like age, proficiency in Elm programming, level of experience with different code editors, and minor disabilities within their characters. The personas were created from observations, an interview with the course Teaching Assistant, and the researcher's experience as a Teaching Assistant in similar courses, and we will use them to evaluate solutions in the ideation phase. We must examine the effectiveness of our solution from the perspective of our respective personas.

Our first persona is John. He is a CS1 student. He is familiar with Haskell, but this is his first experience with Elm. He is proficient in using a variety of text editors and code editors. He has poor vision. Figure 4.1 shows this persona.

Our second persona is Dr. William Smith. He is 45 years old. He holds a Ph.D. in computer science and has been teaching 1XD3 for the past five years. He is an expert in Elm and enjoys teaching online courses. Figure 4.2 shows this persona.

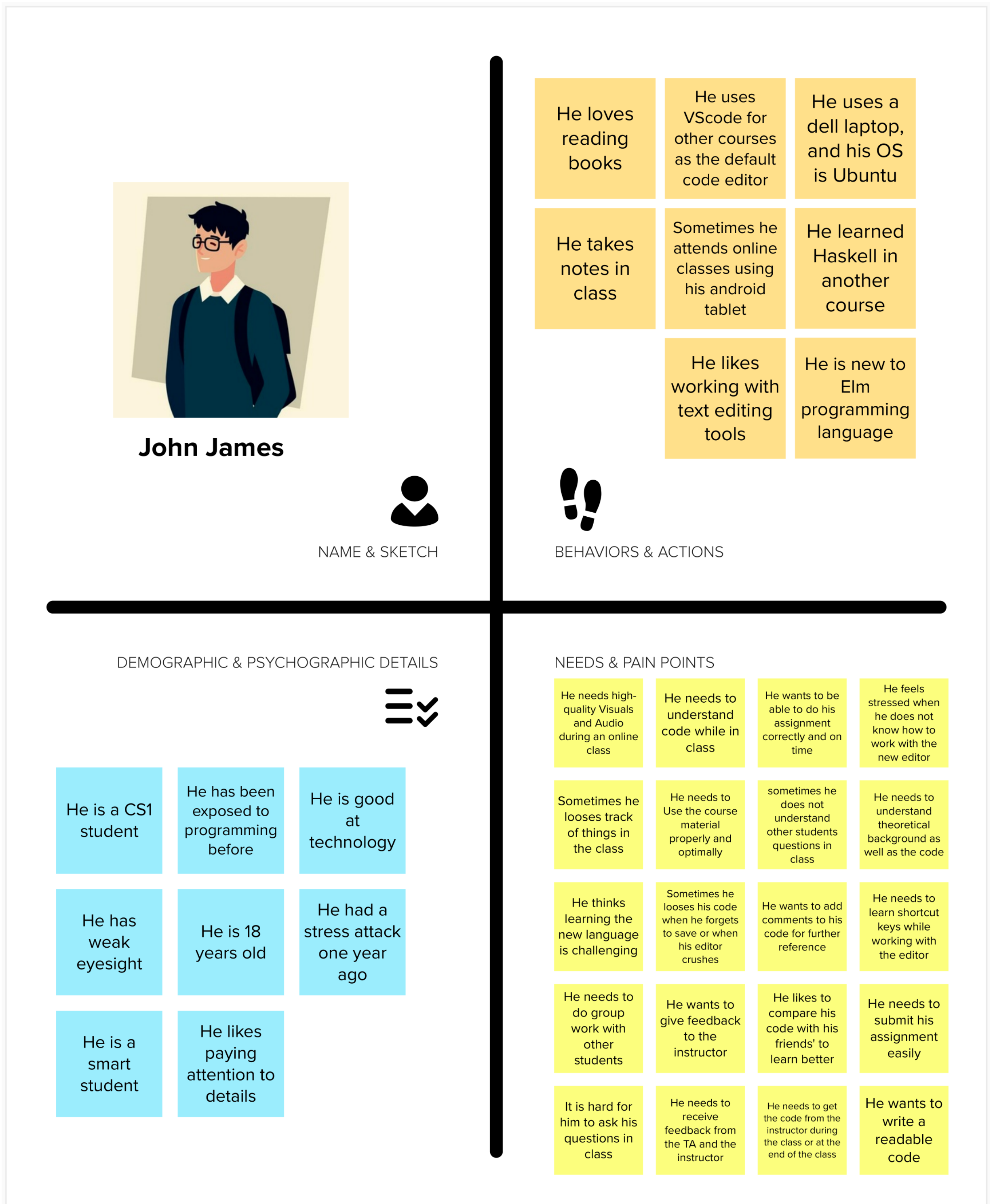

FIGURE 4.1: Student Persona

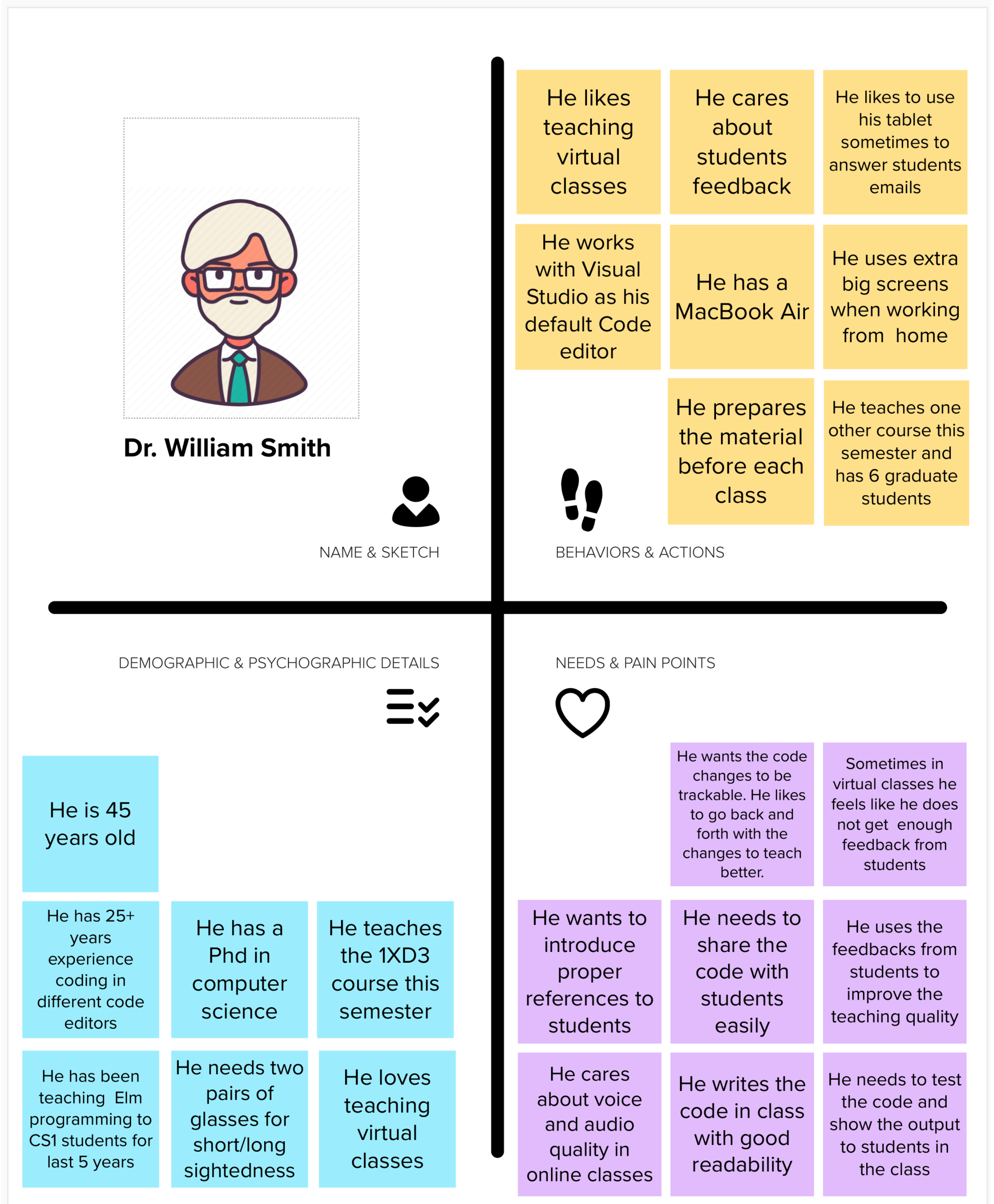

Figure 4.2: Instructor Persona

4.4 Defining the problem

To move from Empathy to Define, data collected from observations must be organized and processed to gain some insight. We will explain how we synthesized and processed our data in this section.

As explained in section 4.2, we collected all the raw data in the form of What-How-Why statements. These statements were drawn from several observation sessions we conducted and were not manipulated in any way. A total of 96 statements were collected. It is very difficult to make sense of this large number of statements directly, but the clustering of ideas helps.

To better understand the data, we clustered it. We chose the term cluster over classify because we did not know the number of classes and their labels. This step aims to identify the most related observations and group them while keeping the more distinct ones in separate clusters.

1. We could put our observations into 12 different categories in the first pass. Code Optimization, Commenting, Copy/Paste/Cut, Debugging, Editing, Deleting, Explaining, Finding, Highlighting, Readability, and Testing. Then we grouped the most relevant ones to be able to synthesize observations from resilient categories. We ended up with five groups. Figure 4.3 shows this grouping and the number of statements in each group.

2. We analyzed the observations in each group, removed duplicates and irrelevant information, and merged similar ideas. As a result, we reduced the observations to 33 meaningful statements that included numerous details.

3. To have the chance of synthesizing points from different groups, once again, we put all 33 statements together. We synthesized the ideas together and used them

to write 21 POVs. The first column of the result tables 4.1 shows the POVs.

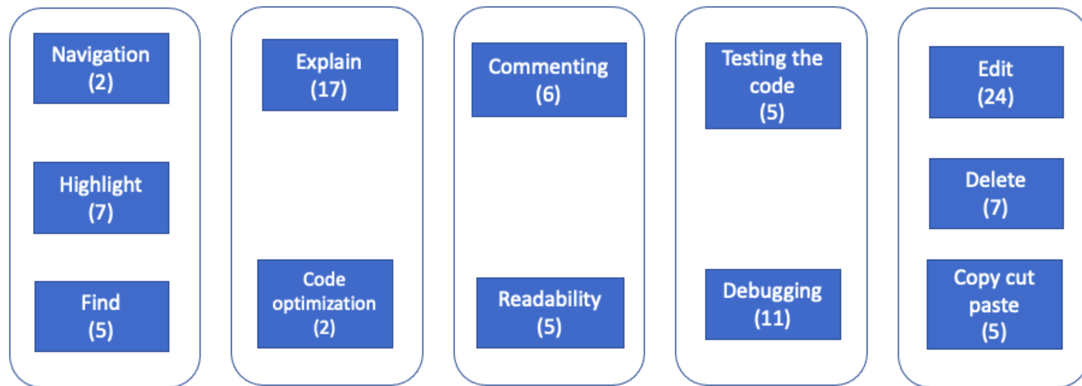

FIGURE 4.3: Grouping Observations

4.5 Ideation and Brainstorming

We needed to write "How Might We" questions for each of the POVs to initiate brainstorming. Asking the correct question is very important because it will open new doors to innovation. As an example, our POV 16 says: "The instructor needs to select specific parts of the code because he wants to attract the student's attention. The student who is easily distracted or confused should identify which part of the code is being discussed." The first question that pops into one's mind as a How Might We (HMW) might be: "How might we help the instructor select the code easily?" but a better question would be, "How might we help the student to know which part of the code is being explained?" The first question will lead to various ideas about how to make selection more efficient, while the second question will open the door to many ideas to help the instructor bring the students' focus on a piece of code.

In order to support a broad brainstorming process, we attempted to provide many questions. We have listed these questions in result tables 4.1-4.7 . Later, we began to provide solutions. During this step, a designer must avoid making judgments and provide as many creative solutions as possible for a given point of view. These solutions can

be expensive, time-consuming, and even appear impossible. Nevertheless, the designer should consider all of them since crazy ideas can trigger better ideas in the future. We did this process by providing the solutions in the third column of our Excel sheet.

The POVs, their corresponding HMW questions, and the brainstorming results are reported in result tables 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7. We will discuss our observations, problem statements and design suggestions in details in Chapter 5.

#	POV	How Might We	Brainstorming
1	The instructor, when coding in the class, needs to review the parsing errors in a timely fashion and navigate to the error line as quickly and easily as possible because class time is limited, and he cannot run the program without correcting the parsing errors.	1. How might we make fixing parsing errors easy for the instructor? 2. How might we help the instructor navigate to the error line easier? 3. How might we help save the class's time concerning fixing parse errors? 4. What if we could eliminate parse errors? 5. What keeps us from eliminating parse errors?	1. Design a projectional editor that prevents parse errors 2. Hire more professionals that don't make parse errors. 3. Train the instructors to avoid parse errors 4. Design a systematic parse error guide that helps the instructor 5. Creating a training app to help instructors get faster in fixing parse errors
2	The Students need to learn how to handle parse errors because otherwise, they will be unable to debug the syntax errors and move on to the next step of learning, which is to understand computer science concepts and be creative.	1. In what ways might we help students learn to handle parse errors? 2. How might we eliminate the parse error barrier from students' learning experience?	1. Give them instructions, including common parse errors and their meaning and common fixes. 2. Spend the specific time of class to create parse errors and solve them. 3. Teach a small example of a parser in class 4. Design a projectional editor that prevents parse errors
3	Students and instructors need to regularly move the cursor using mouse and keyboard because they need to edit the code. For example, they navigate between the parenthesis or brackets. People with mobility difficulties may find this action difficult.	1. How might we make navigation easier in our editor? 2. How might we help users with mobility problems to navigate easier using our editor? 3. In what ways can we make navigating quicker?	1. Design navigation using eyesight 2. Design navigation using voice. For example, when the users want to go to a line, they can read a part of that out loud, and the cursor goes to that code. 3. Make navigation with the keyboard faster. (People with mobility limitations find working with mouse difficult.) 4. Use holes for editable parts of the code.

Table 4.1: Results

#	POV	How Might We	Brainstorming
4	Students and instructors need to do indentation quickly and easily because indentation increases the code readability and prevents parsing errors.	1. How might we help the user do indentation easier? 2. How might we enhance indentation to improve code readability? 3. How might we prevent parsing errors caused by indentation?	1. Make indentation automatic 2. design a button. Then, when a piece of code is selected, and that button is pushed, the indentation problems of that code get fixed. 3. Make the text draggable so the user can easily drag it to where it should be. 4. Create templates for code that follow correct indentation.
5	Students and instructors need to memorize the names of the defined functions and variables before they reuse them. Because if they do not remember them, they may have to look for them in the code or make typographical errors that lead to parsing errors.	1. How might we help users remember the function and variable names easier? 2. In what ways can we remove the need for the user to keep names and functions in his mind? 3. How might we remove the chance of typographical errors in the code?	1. Keep a list of all created variables inside of the coding windows. 2. When the user types a name that is partially close to a name that already exists in the code, check the spelling and underline red if it is typed incorrectly. 3. When the user is typing a partially close name to a name that already exists in the code, show the similar name in a tooltip. 4. Let the users choose from a list of existing names when they need to.
6	The instructor who codes in the class and has experience with many code editors needs to write and edit his code easily and fast in the class.	1. In what ways can we make editing code quick and easy? 2. In what ways can we help a user to learn editing functionalities easy?	1. Make the editing experience very close to frequently used editors. (Design for experts) 2. Give visual signifiers to the users, so they know the functionalities. 3. Create an instructional video.

TABLE 4.2: Results-continue

#	POV	How Might We	Brainstorming
7	The coder needs to search within the code because navigation gets difficult when the code is long.	1. In what ways can we help the user search within the code? 2. How might we make the navigation in code easier?	1. Give the user a search option that is always on the top side of the editor. 2. Support partial search. 3. When the user double-clicks on a name, other occurrences get highlighted. 4. give the user a find option that activates by key combinations.
8	The coder needs to comment in the code because it helps him add extra information to the code.	1. How might we help the instructor leave comments in the code clearer? 2. How might we help the user add extra notes to the code? 3. In what ways can we keep extra information assigned to the code?	1. give the option to the user to choose the comment color. 2. Specify comments with a sign on the screen. 3. The comments appear on the screen bold or bigger in size. 4. give him a note function that is independent of code. And only assigned to a specific location in code.
9	The user needs to do things in the order he wants because that is the way to support his creativity and problem-solving skills.	1. How might we support the users' creativity? 2. In what ways can we help the user be creative? 3. How might we let the user do things in arbitrary order?	(This item can happen in normal code editors easily, and it is not a challenge. But in projectional editors, it is a challenge since users' actions are selected from a menu.)Support dummy names. For example, when the user wants to use a function before defining it. It can choose a dummy name and then create the function. Note: solutions for this requirement in the framework of a projectional editor need more observation to find specific cases.

TABLE 4.3: Results-continue

#	POV	How Might We	Brainstorming
10	The instructor and students need to see the output of the code when it is running because they should see the effect of the changes they have made.	1. How might we make the observation of output easier? 2. In what ways might we help the user see the output window easier? 3. How can we help the user see the effect of his changes easier on the output?	1. Design an output screen within the editor. 2. Make the output screen adjustable in size. 3. Include zoom in/out options. 4. design an output screen that shows the code changes immediately. Meaning running, for example, every 10 seconds!
11	The instructor needs to zoom in and out to make the code visually clear because the students must see the code clearly all the time in class.	1. How might we support zoom in and out? 2. In what ways can we change the size of the code? 3. how might we make the code more visible to the students?	1. provide a zoom in/out option for the users. 2. Provide a size and font option for the user. 3. Give a menu to the user to choose the code color! 4. Give a menu to choose the background color. 5. Add an option for dark background
12	The students need to view the output screen clearly. They also need to understand the scale and coordinates of the output screen.	1. How might we help the user see the output screen clearly? 2. In what ways we can support the student to understand coordinates of the output screen? 3. In what ways can we help the student work with output screen better?	1. Add a grid on the output screen with some numbers that can indicate the coordinates 2. Add an option to show coordinates 3. Give him a magnifier tool to zoom on required parts he wants 4. Make all items on the output screen bigger when the user adjusts the edges of the output window.

Table 4.4: Results-continue

#	POV	How Might We	Brainstorming
13	Students and instructors need to know the model values while the code is running because it helps them understand the semantic errors that do not appear obvious on the screen.	1. How might we help the user know the model values? 2. In what ways we can show the model values to the user? 3. In what ways we can support the user with semantic errors?	1. Give an option to the users to display model values on the output screen whenever they want. (Runs the Debug.toString always but does not display until the user wants) 2. Dedicate a part in User Interface (UI) to show the model values when code runs. 3. create a list of common semantic errors. Then, if the system found any of them in the code, it should lead the user.
14	The instructor who codes alone or with his students in class needs to run the code quickly and easily since this action frequently occurs during coding and should be straightforward.	1. How might we design a way to run a code suitable for in and out of class. 2. In what ways can we design an affordance for running the code which is both appropriate in and out of class?	1. Design a big button with a very bright color that attracts attention. 2. Use a shortcut key to run the code, so it is very fast. 3. Run the code automatically if the user does not change anything for 5 seconds! 4. Run the code automatically when the user saves the code. 5. design a vocal sensor that runs the code whenever the user says run!
15	The student who attends the online class needs to know when the instructor runs the code because otherwise, he may become confused.	1. How might we help the student to be aware of the code running? 2. How might we distinguish between editing and running modes in the editor?	1. Ask the instructor to say the code is running each time he runs the code. 2. Associate a voice saying running when the instructor runs the code. 3. Create an animation for when the button is pushed and change the color as long as the code is in running mode. 4. Draw a big rectangle around the output window when the code is running.

TABLE 4.5: Results-continue

#	POV	How Might We	Brainstorming
16	The instructor needs to select specific parts of the code because he wants to attract the student's attention to the code. The student who is easily distracted or confused should be able to identify which part of the code is being discussed.	1. How might we help the instructor select code more efficiently? 2. In what ways can we help the instructor draw students' attention to a specific part of the code? 3. In what ways can we show the student which part of the code is being explained? 4. What if we give a highlighting tool to the instructor that works better than the current selection function?	1. Give the option to the instructor to select code in a different color. 2. Give the option of explaining mode to the instructor, so in this mode, when the instructor selects a structure, draw a frame around the whole structure. 3. Give the instructor a highlighting tool with different shades so selecting and highlighting have different functions. 4. Let the instructor highlight a part of the code and leave it like that. So he can find that part of the code easily.
17	The instructor needs to point to certain parts of the output screen when he is explaining because he wishes to draw the students' attention to that particular part of the screen.	1. How might we help the instructor to point to output objects easily? 2. How might we make pointing done by the instructor more visible to the students?	1. Give the instructor a selection of cursor shapes to choose from. 2. Give the instructor the option to customize the cursor shape in color and size.
18	The student who easily gets distracted or has some form of visual impairment must be able to see exactly where the instructor points in order to better understand the lesson and avoid confusion.	1. How might we make pointing done by the instructor more visible to the students? 2. How might we support the visually impaired students in the class?	Solutions from 13 apply. But the animations should be slow so with poor video share connection the animation is still visible. And for the student who easily get distracted the color and size can draw attention.

TABLE 4.6: Results-continue

#	POV	How Might We	Brainstorming
19	The students need to know whether a key is pressed or held by the instructor when the program is in running mode because the keystrokes are one of the important actions to test the code.	1. In what ways can we inform the student with the keystrokes done by the instructor? 2. How might we make keystrokes visible on the screen?	1. Associate a voice saying the name of keys being pressed. 2. Show the name of keys being pressed to the user on the screen.
20	The instructor who teaches computer science concepts while coding in class needs to have access to online resources because extra resources support him to explain the concepts better.	1. how might we provide access to online resources for the user? 2. In what ways can we make switching screens easy and quick? 3. What if we have an integrated environment that gives access without switching?	1. Provide a link to frequently used resources. 2. Design a web application. 3. Design an integrated environment that has access to other resources.
21	The instructor needs to share the code easily with the students because the students need to practice working with it, edit it, or complete it.	1. How might we help the instructor share the code? 2. In what ways can the student reach the code?	1. Support select all and let the users copy the whole text at once. Then, they can use other tools for sharing the code. 2. Give the option to the instructor to share the code with a list of users within the editor. (Every user has a username, and he has many working spaces. We can, by default, copy the code to the workspace for a specific session. or the users' default chosen space)

Table 4.7: Results-continue

Chapter 5

Discussion And Design

When designing software applications using Design Thinking (DT), the design thinker can empathize by observing the user's interaction with a legacy system. A legacy system is software that has been developed over a period using outdated techniques but still performs critical functions within an organization. This system will likely be difficult to maintain or modify, and it requires substantial modernization. The system's modernization involves significant changes, such as implementing new and relevant functional requirements, changing the software architecture, or migrating to a new platform. [7]

In such a project, we face two types of requirements:

- *Known requirements:* The features that already exist in the legacy system are the known requirements, and you can collect them from an Functional Requirements Specifications (FRS) document. These requirements have already been recognized and are somehow met within the current design. Approaching known requirements with DT, we should first understand the need (empathy) and judge if it needs reframing or not, which means checking if we are solving the correct problem or need to change the definition of the problem. If the definition is correct, we can move to the ideation step. Sometimes the solution provided by the legacy system is efficient and widely accepted; therefore, we might want to keep it the way it is.

- *Hidden requirements:* These features are hidden requirements that the customer cannot recognize and can be revealed using empathy. These requirements can appear when we do User Centered Design (UCD), and we can start with asking questions like "What is needed by a specific persona that a general user does not need?" For example, in this study, we can ask what features in a code editor are required by an instructor and not an ordinary user.

As expected, our research is conducted on a legacy system. Our legacy system is a web application designed and implemented by our research team to create an accessible Elm coding platform for everybody. Currently, we use this application to teach coding to kids as a part of our outreach program. The instructor uses the same tool to teach Elm programming to Computer Science students in their first year of university due to its accessibility for novice users. Even though our legacy system is not written with obsolete methods, it depends on unmaintained libraries. It originated as an undergraduate course project and has become difficult to evolve.

To address our second research question, we need to explore whether there is enough evidence to suggest that projectional editors can be useful in the teaching of coding to students in CS1. In case enough evidence is found, we add requirements specific to projectional editors.

5.1 Projectional Editor or Not?

We identified many interesting pain points during our observations. Several of them support the idea that a projectional editor can facilitate the teaching of coding to first-year students.

- **POV1:** The instructor, when coding in the class, needs to review the parsing errors in a timely fashion and navigate to the error line as quickly and easily as

possible because class time is limited, and they cannot run the program without correcting the parsing errors.

- **POV2:** The students need to learn how to handle parse errors because otherwise they will be unable to debug the syntax errors and move on to the next step of learning, which is to understand computer science concepts and be creative.

- **POV3:** Students and instructors need to regularly move the cursor using mouse and keyboard because they need to edit the code. For example, they navigate between the parentheses or brackets. People with mobility difficulties may find this action difficult.

- **POV4:** Students and instructors need to do indentation quickly and easily because indentation increases the code readability and prevents parsing errors.

- **POV5:** Students and instructors need to memorize the names of the defined functions and variables before they reuse them. Because if they do not remember them, they may have to look for them in the code or make typographical errors that lead to parsing errors.

Based on our observations, a lot of time in a classroom is spent fixing parse errors. A good editor offers ways to simplify this task. The legacy system supports easy access to the error line by providing a link. But what if we were to prevent syntax errors? Projectional editors are widely acknowledged to be a solution for avoiding syntax errors. As explained in chapter 2, projectional editing is a technique to manipulate the abstract syntax tree directly, bypassing the parser. Obviously, taking advantage of such a framework as our base design platform can remove our biggest challenge and create scope for other mini solutions. So as our main design decision, first, we decided to create a projectional editor, and all of our other discussions are under the influence of our first decision. Projectional editors help eliminate errors in two ways. First, they provide a template

for each structure, and the coder can change the code only with respect to the template. Second, when a hole needs to be filled by the coder, the projectional editor gives hints for easier navigation. Additionally when the user does not fill a hole, the projectional editor fills it with appropriate default values to keep it parsable all the time. (POV1)

Using a projectional editor in the classroom, the instructor and students would never face a parse error again. Therefore, several minutes of every class would be saved to be spent on deeper computer science subjects. Since students would still at some point need to use a conventional editor, they would need to understand syntax errors. Therefore, we can never omit that from their learning experience. But do they need to be exposed to parse errors in every course?

After discussing this with the instructor, we asked if there are other ways to teach parsing. Can using a projectional editor save enough time for the students to write a parser for a mini-language or for regular expressions? Thereby, the students would learn the concept in better depth. Additionally, leveraging the class's limited time to teach more computer science concepts would be of more value to the students than spending time fixing syntax errors. (POV2)

Students benefit from syntax error-free coding in other ways as well. For example, students have difficulty fixing syntax errors when learning a new language, and completing assignments is challenging and time-consuming. In addition, the students feel exhausted after too much time trying to fix an error and may feel discouraged. If syntax errors are removed, they may feel more interested in the subject and learn better. CS1 emphasizes teaching computer science concepts and motivating creativity in students, so removing syntax errors can lead to a better learning experience for students. (POV2)

On the other hand, students are exposed to syntax errors in many other courses

while doing projects. Therefore, they will learn how to handle them eventually. We conclude that the benefits of using a projectional editor outweigh its disadvantages overall. (POV2)

Based on our POV3, navigation in code is also a challenge. Specifically, it can be a big challenge for our student persona John. They have mobility limitations and finds working with a mouse a bit challenging. A projectional editor creates the skeleton of a language structure when the user triggers it. For example, if the user triggers an "if statement"[1] , the whole skeleton will be shaped:

```
if [conditionHole] then
    [expressionHole1]
else
    [expressionHole2]
```

To make navigation easier, we use arrow keys on the keyboard to navigate to a hole. The right and left arrow keys should move the cursor between the children of the same parent, and up and down arrows should move the cursor to parent and child of a node, respectively (POV3). The template created by the projectional editor should also follow correct indentation to avoid parse errors and follow readability principles. This way, it eliminates the users' effort to do correct indentation in the code. (POV4)

Another reason to support the idea of a projectional editor is the challenge with memorizing variable and function names. According to our POV5, when the code is big and the coder defines many variables and functions, it gets difficult for them to remember all the names. Based on our observations, the coder uses the find function to find a name that they remember partially. This method is useful when the coder remembers some part of the name and the system's find function supports partial search

[1]The user can trigger an If statement by typing "i" for example, This study does not cover the projectional editor's detailed behavior of creating language structures because this level of detail could not be derived from our observations.

functionality. However, when the user does not remember the name partially, they must navigate through the whole code to find the required name. Projectional editors can provide a straightforward solution to this problem. When users navigate to a hole, the system provides options for them to choose from. When the coder needs to use a name, whether a function or variable, they can navigate to a hole and choose from the drop-down menu. The menu only shows suitable names based on their type. Following this method eliminates the need to remember names and prevents parse errors due to typos or type errors.

Based on the expected efficiency gains due to the analyzed POVs, we can positively answer RQ2. Therefore, subsequent design decisions about editing are made with respect to a projectional editor.

5.2 General Editing Requirements

The second group of POVs offers insight into the required editing features:

- **POV6:** The instructor who live codes in class and has experience with many code editors needs to write and edit their code easily and quickly in class.
- **POV7:** The coder needs to search within the code because navigation becomes difficult when the code is long.
- **POV8:** The coder needs to comment in the code because it helps them add extra information to the code.
- **POV9:** The user needs to do things in the order they come to mind because that reduces cognitive load, freeing cognitive resources for creativity and problem-solving skills.

Editing in a projectional editor is necessarily different from editing in a parser-based editor. This section discusses the design decisions we made regarding editing actions in a projectional editor.

The most critical experience in a code editor is editing. The instructor codes most of the time while the students watch. Although editing is a major activity in the class, it is a means for the instructor to teach more sophisticated concepts. It should, therefore, not be a complicated or time-consuming task.

Both instructors and students benefit from simplifying the editing process. With an easy editing experience, a instructor can edit faster and spend more time explaining computer science concepts in class. Furthermore, students will not be confused and distracted by complicated editing processes. If students can do their assignments on the same platform, they will feel more comfortable editing code.

Obviously, the standard text editing features such as select, copy, cut, paste, delete, undo, redo, find, and replace should be available in our code editing environment.

Our study participants are familiar with many text and code editors, so one approach to ease the editing experience would be to make it as similar as possible to their previous code editing experiences. The users can more easily learn our system if the affordances are familiar to them. To begin with, we use the common shortcut keys. The system must behave as closely as possible to expected behaviors but within the paradigm of a projectional editor. In addition, to conform to Norman's first principle of design and Hansen's redundancy rule that there should be more than one way of doing something, we provide an editing menu that includes these features. We do not have evidence to support choosing one option over another in designing such a menu, so we suggest a simple drop-down menu that displays the functions and their shortcut keys. Based on feedback from users on prototype 0, we can improve this menu in subsequent DT

iterations.

5.2.1 Selecting

Observations of the instructor included the following selection behaviors:

1. Holding the mouse over a line, multiple lines, or a word to select it. It happens from top to bottom and vice versa.
2. Double-clicking on a word to select it.
3. Clicking three times on a line to have the whole line selected.
4. Pressing Ctrl+A to select all the code.

The selection unit is the most significant factor in selection. Since the code grows along the Abstract Syntax Tree in a projectional editor, selection can be made on subtrees.

In our projectional editor, selection by mouse will work as follows:

- Double clicking selects the smallest subtree containing the click point. (Example: If users click inside a variable, it will be selected. If they click inside the 'if' in an if expression, the whole structure will be selected.)
- Triple clicking goes one level up toward the root. It selects all children having the same parent as the smallest subtree that can be selected by double-clicking.
- Quadrouple clicking goes two levels up toward the root. (We want to include this in our prototype to test the user feedback, since it is non-standard, but consistent with the other actions.)
- ctrl+a selects the program tree.

Our projectional editor allows the user to select by keyboard using arrow keys too. Wherever the cursor is, right and left arrows move the selection between children of the same parent, and up and down arrows pass the tree in-depth, meaning moving from a branch to its parent or its first child.

5.2.2 Deleting

The following deletion patterns were observed:

1. selecting the code, pressing delete on the keyboard.
2. selecting the code, retyping.
3. putting the cursor after what needs to be deleted and pressing delete on the keyboard multiple times.

The above patterns are all composed of selecting and deleting, and they follow the rules of selection mentioned above. Anything that can be selected can be deleted. When a unit is deleted, it is replaced by a hole of the same type. If the user wants to run the code without filling the hole, then one of the following applies:

1. We use our default values that match the hole's type. For example, with an integer, we fill it with 0, and with a string, we fill it with an empty string.
2. If the user defines a new data type with multiple constructors, the system will fill the hole with the first constructor.

A user can select a unit and begin typing, and the system will attempt to match it with a subtree in the clipboard or a new structure, definition, or constructor. If nothing matches, the action is ignored. This function needs to be defined in greater detail during prototyping because the observations did not show evidence to guide the solution.

5.2.3 Copying and Cutting

Whatever can be selected can also be copied or cut. There is no need for clipboards in parser-based code editors because the users can paste their code anywhere and comment on it. Such action, however, has complications in project-based editors.

Copying and cutting in our code editor can be done using the menu or by pressing Ctrl+C and Ctrl+X, respectively. Copying will not affect the code itself, but cutting deletes the selected subtree and follows the same rules as deleting. We suggest a clipboard that keeps recent copied/cut items, including at least the last cut/copied subtree of each type, and perhaps multiple such most recently cut/copied subtrees.

5.2.4 Pasting

When a user pastes over a selected hole, one of the following applies:

1. If only one item in the clipboard matches by type, it will be pasted.
2. If there is more than one item in the clipboard matching by type, the user can select one.
3. Otherwise, the system notifies the user that there is no suitable subtree in the clipboard to paste.

If the user wants to replace a selected unit by pasting, one of the following applies:

4. If one of the actions 1 or 2 is executable, the system removes the existing code and does as above.
5. Otherwise, it gives a message to the user and ignores the action.

5.2.5 Undo and Redo

We recommend keeping two linked lists of states for undo and redo. Undoing pushes the current state to the redo list and fetches the most recent state from the undo list. And redo fetches the most recent item in the redo list to the current state and pushes the current state to undo list. The user can activate undo and redo functions using the common shortcut keys, Ctrl+Z and Ctrl+Y/Ctrl+Shift+Z, respectively.

5.2.6 Find and Replace

When the code grows, navigation becomes harder, so the coder needs an easy way to navigate through it. The find function helps in this regard. During our observation, we observed the instructor using the find function in two ways:

1. Pressing Ctrl+F then typing the word then enter.

2. Selecting a piece of code, Ctrl+F, then enter.

The system should activate the find function when the user presses Ctrl+F on the keyboard. A small window appears on the left side of the screen with a text box. There should be two small arrows to help navigate through found options. Our search feature supports partial searches, so an instructor can use it if they recall only part of the name or some part in the middle. Additionally, this supports POV5 by saving the programmer from having to remember names.

We support both ways that the user expects our find function to work. The user can activate the find function and type in it. Also, if they select a subtree and press Ctrl+F, the system will search for the subtree. Then all the occurrences of that subtree should be highlighted through the code and be navigable using the small arrows.

The legacy system does not support the replace feature. However, it has been noted that instructors may need to insert replacements quickly, so we suggest adding this

function to the editing tools. A user may replace a subtree with another subtree of the same type. A user may also rename a variable, and all instances of that variable in the same scope should be renamed.

5.2.7 Commenting

We observed the following commenting behaviors from the instructor:

1. The instructor comments about something that is not a code explanation, like FIXME or TODO.
2. The instructor comments a single line or a piece of code (multiple lines) to exclude it from running temporarily.
3. The instructor adds a comment about the explanation of code for students to refer to in the future.
4. The instructor takes temporary notes using comments. Like a pre-defined function name (possibly with signature) that they copied from the Elm package guide.

In a projectional editor, every comment must be attached to a subtree. The user may create a comment for a specific subtree by selecting it and typing double dashes. With this action addition, items 1, 3, and 4 can be handled in the projectonal editor. If the user selects a subtree and types double dash, first the system comments the current code and then creates a hole following delete rules.

5.2.8 Out of Order Editing Habits

Many developers have a habit of doing things out of order. There are two main reasons for this based on our observations:

1. They think they will forget to complete an action later, so they wants to perform it now—even though it is out of order—to prevent a parse error. Like: Immediately typing the closing parenthesis after opening it

2. They have to record something immediately because they are thinking ahead of the code. So, for example, a coder might use a function before defining it.

In the first case, the coder has been left to remember to do an action later. A major advantage of projectional editors is that they build the skeleton of a language structure and prevent such errors. For example, when the coder types "(" the system automatically puts the ")" after it and gives the code a hole between parenthesis to fill. Or, when the coder wants to create a "let $\cdots$ in $\cdots$" structure, the editor gives them the skeleton, so the coder does not need to create it themselves and have to worry about forgetting something.

The second group is about how the developer thinks, and we might want to let them do things in the order they see fit. Therefore, we must consider the developer's behavior in the first place. Our observation was limited and only revealed one habit in this category. For the exact observation related to using a function without defining it, we can give the user the chance to use a dummy function or variable. Therefore, we will add a dummy to the list of available options displayed to the coder to meet this need. Dummy is a hole of the correct type that the users will fill later. Further research and observation are needed to address more items from this group. We recommend that the team continues noting related observations to find similar requirements and include them in future iterations based on priority.

5.3 The User Interface

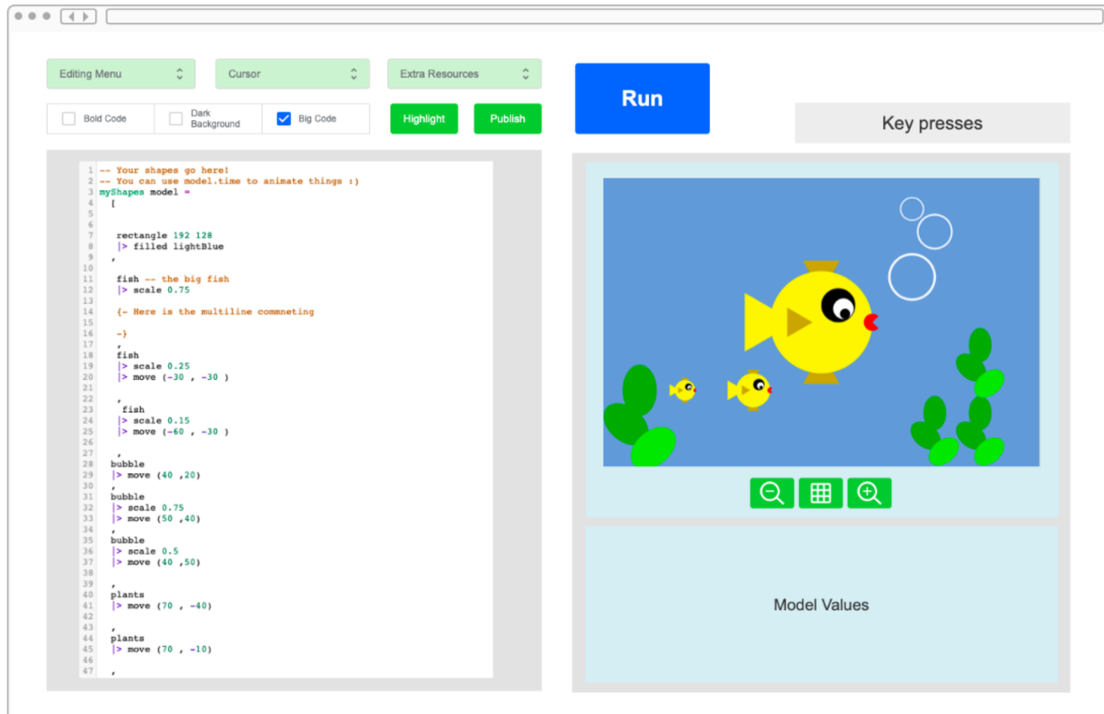

Figure 5.1: Suggested User Interface

Users and software applications are linked by the User Interface (UI). A software's UI includes all the navigation and feedback components required to navigate the application and make decisions. It refers to how an application interacts with users and presents information. Informational elements, navigational controls, and containers are all part of the User Interface Design. Also, buttons, lists, toggles, icons, tags, and more play a huge role. We present our design decisions concerning the editor's user interface in this section based upon our discovered POVs.

- **POV10:** The instructor and students need to see the output of the code when it is running because they should see the effect of the changes they have made.

- **POV11:** The instructor needs to zoom in and out to make the code visually clear because the students must see the code clearly.
- **POV12:** The students need to view the output screen clearly. They also need to understand the scale and coordinates of the output screen.
- **POV13:** Students and instructors need to know the model values while the code is running because they help them understand the semantic errors that do not appear obvious on the screen.
- **POV14:** The instructor who codes alone or with their students in class needs to run the code quickly and easily since this action frequently occurs during coding and should be straightforward.
- **POV15:** The student who attends an online class needs to know when the instructor runs the code because otherwise, they may become confused.

On the editor's main page, the screen should be divided into two main sections vertically to accommodate both code and output. The default position of the dividing edge is in the middle of the screen vertically. Window sizes should be adjusted by a dragging affordance that lets the users adjust the window size. (POV10)

The left section is where users can enter the code. To improve the visibility of the code during online courses and assist visually impaired students, accessibility options should be available. A setting should allow users to adjust the size of the code. There should be at least two sizes available for changing the size of the code, an option for making the code bold, and an option for changing to a dark background. The menu should always appear at the top of the code window. (POV11)

The right side of the screen should be divided into two sections horizontally. At the top, there should be a larger area to display the output of the code. The user will see the

results in this section immediately and easily whenever they run their code. Additionally, this window should have a zoom-in and zoom-out control, including a percentage option. Finally, There should be an option to add a grid view to the output screen. This helps the students to navigate better in the output screen. (POV10, POV12)

We designed a section at the bottom right of the screen to display the model values when the program is in running mode. Using this section will facilitate debugging semantic errors and allow the coder to be aware of the changes to the model while simultaneously observing the output. As the legacy system did not support the display of model values, the instructor had to add code (Debug.toString model) to display the values on the output screen. It was not only an extra effort for the programmer, but it also consumed some of the screen space. (POV13)

A blue rectangular three-dimensional run button should appear at the top of the output window as an affordance to run the code easily (POV14). The run button should be indicated with a button signifier to show a pushable button and an animation to demonstrate when it is actually clicked on. The animation is the system's feedback (Principle #2) in response to pushing the button. All the surfaces of the button should be clickable and should allow the code to run. When the button is pressed, its color should change to gray. The button remains gray as long as the code is running, and no changes have been made to it. It changes to blue as soon as a change is made to the code. In this manner, even if the student becomes distracted and misses the moment the instructor runs the program, they will know that the program is running by looking at the button. (POV15)

By putting a button on the screen for running the code we adhere to Norman's visibility principle (Principle #1). Based on Hansen's principle #12 (redundancy) we should have more than one means for an action. Therefore we use the shortcut key F5 to perform the run action too. So that pressing the F5 key is equivalent to pressing the

run button using the mouse (POV14). It will activate the animation, change the color of the button, and run the code. We chose F5 to run the code since it is a common way of running the code in many popular code editors. Since our personas both have experience with several code editors, it makes sense to use the familiar shortcut keys to do specific actions. Also, this adheres to Hensen's 7th principle of design (muscle memory).

5.4 Teaching Functionalities

Several of the POVs emphasize classroom experiences rather than the experience of a coding practitioner in general. These POVs reveal the hidden needs that emerge when we consider the special needs of our personas compared to the legacy system users.

- **POV16:** The instructor needs to select specific parts of the code because they want to attract the student's attention to the code. The student who is easily distracted or confused should be able to identify which part of the code is being discussed.

- **POV17:** The instructor needs to point to certain parts of the output screen when they are explaining because they wish to draw the students' attention to that particular part of the screen.

- **POV18:** The student who easily gets distracted or has some form of visual impairment must be able to see exactly where the instructor points in order to better understand the lesson and avoid confusion.

- **POV19:** The students need to know whether a key is pressed or held by the instructor when the program is in running mode because the keystrokes are one of the important actions to test the code.

- **POV20:** The instructor who teaches computer science concepts while coding in class needs to have access to online resources because extra resources support them to explain the concepts better.
- **POV21:** The instructor needs to share the code easily with the students because the students need to practice working with it, edit it, or complete it.

To meet the needs grouped as teaching specific ones, we intend to provide a teaching toolbox within our code editor. This toolbox includes specific tools that the instructor needs based on our observations.

First of all, the instructor needs to press or hold a key on the keyboard in order to test the code. Also, sometimes they need to click on a part of the output to test a functionality. During our observation, we recorded multiple questions from students regarding the keystrokes or mouse clicks, as the instructor tested the output. Because the students in an online (or any) class cannot see the instructor's keyboard, they are not aware of the instructor's actions using the mouse and keyboard and since these actions are essential while testing the code, the students can easily get confused (POV19). We want to design a mechanism to show keystrokes and mouse clicks on the screen. We recommend having a small window at the top right next to the run button that indicates the keystrokes and mouse clicks made by the user when the environment is in run mode.

Secondly, another observation shows that the instructor frequently points to various objects in the output window when explaining. Pointing assists the instructor in explaining things clearly. Following the mouse movements is necessary for the student to avoid confusion. All students, including those with some visual impairments, should be able to see the mouse movement (POV17). As a result, we recommend that our editor provide an option for a clearer cursor shape. The suggested cursor shape is an arrow in dark color and bigger size which make it more clear for the students specifically in

a screen-sharing situation. Additionally, to adhere to Norman's feedback principle and respecting POV19, we want to associate an animation to the cursor for mouse clicks. When a click happens in running mode on the output screen, a few lines surrounding the arrow appear. This animation lasts for 2 seconds to compensate any possible connection lag in an online class or video compression artifacts.

Another observation shows that the instructor repeatedly selects certain parts of the code while explaining because they wish to emphasize the part being explained (POV16). This observation provides a good example of how DT can reveal hidden requirements by asking the correct How Might We (HMW) questions. In this example, we discovered the hidden requirement of highlighting for pointing or highlighting for attracting students' attention by asking this question. "Why does the instructor need to highlight the text?" The incorrect question would be, "How can we make it easier for instructors to select text?" This may lead to various solutions, including providing a selection tool, while the actual point is the need for drawing students' attention to some part of the code.

As implemented in the legacy system, selecting the code places a navy-blue highlight around it, which may attract the student's attention. Although selecting does the job in this system, a need for a highlighting tool is not adequately met. The blue highlight caused by selecting makes the code difficult to read. Moreover, any click or keypress will remove the highlighting, so the instructor cannot maintain the highlighting and continue coding. In this regard, we suggest that our editor provide a tool that can meet the needs of instructors.

We suggest a highlighting tool that lets the users highlight any portion of the code that they want. These highlights are permanent in our environment and do not disappear by clicking or key presses. The user can highlight in different colors selected from a menu, but these colors should have enough contrast with the colors used in Elm syntax highlighting. The users can remove the highlights by selecting the text and choosing

"No highlight" from the menu.

Additionally, some observations identified the instructor's need to access other tools and teaching resources. Coding is not the only aspect of teaching programming. Instead, the students should comprehend computer science concepts in-depth (such as state diagrams from PALDraw, a visualization/design tool developed in our research group). Our research group has various learning tools that an instructor can utilize to teach computer science concepts in more depth. Additionally, a coder cannot recall all the function names and inputs provided in a language package. They may regularly use the Elm package website to check for predefined functions, so they require easy access to these packages to locate what they require (POV20). Therefore, easy access to extra learning tools and the Elm package manager is a must in a classroom experience. On the other hand, observing the instructor switching between different tools and resources allows students to observe how a professional finds their way through different resources. We design a drop-down list that includes links to resources that a instructor needs [2] . Using this list, they can find their desired resources easily and use them in class. As our code editor is a web application and the other tools are also on the web, clicking on the link opens a new tab.

Finally, our observations suggested that the instructor frequently shares the code with students. As the legacy system does not support this functionality within the editor, the instructor must select the whole code and paste it into a communication tool like an email or a chatbox. It is inefficient to do this as the code can easily be changed inadvertently and become unparsable. Our suggestion is to provide the instructor with a publish button to share the code with students internally within the code editor. We must define roles for users to handle such tasks. Figures 5.1 and 5.2 show our suggested design for the discussed user interface.

[2] The links being displayed in this drop-down list are controlled by the system administrator or instructional designer and will change based on users' roles.

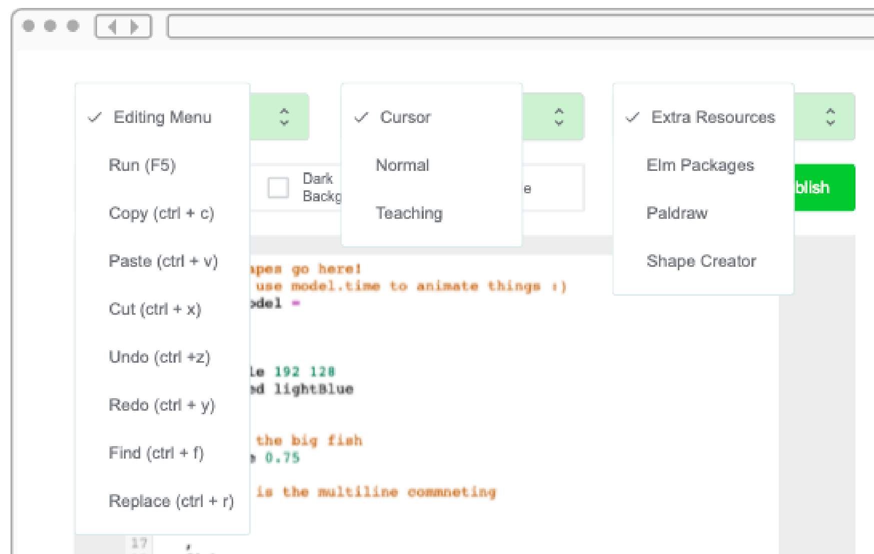

FIGURE 5.2: Suggested Menu

Chapter 6

Conclusion

Using systematic observation inspired by ethnographers, we provide a detailed design for a prototype of an enhanced code editor specifically designed for teaching purposes. Our observations were from screen recordings of classes held virtually due to Covid-19.

6.1 Research Questions

RQ 1: Can observation of a legacy system be used to design a novel code editor specifically used in live coding by an instructor?

Yes. Chapter 4 reports our observations, insights, problem statements, and proposed solutions. Chapter 5 reports the final design decisions.

RQ 2: What features should an editor designed for teaching have?

Based on our observations, we built personas to guide the design. For example we observed frequent use of copying and pasting, undoing and redoing, using common shortcuts. It is important that editors implement these actions in the expected way. On the other hand, frequent use of highlighting had multiple purposes, and an editor designed specifically for teaching can better serve these purposes. Chapter 4 traces 71

possible design decisions back to the original observations, explaining them in terms of the personas.

RQ 3: Is there evidence supporting the usefulness of a Projectional Editor by the instructor in teaching coding?

Yes. Section 5.1 explains how 5 of our Point of View (POV)s reveal pain-points that can be mitigated by using a projectional editor.

RQ 4: What kind of requirements can we get from observing a legacy system? There are two kinds of requirements which can be inferred from observing a legacy system. There are requirements that are not addressed by the current system which can only be inferred from observations of the users' difficulties in using the legacy system for specific tasks. There are also requirements which are met by the current system by using existing features in creative ways. These requirements are likely not to be met if the underlying features are implemented differently in a new system.

6.2 Next Steps

The next steps continuing this project should be:

1. Building the prototype
2. Testing it with real users (instructors and teaching assistants)
3. Testing it from the personas' perspectives
4. Gathering feedback
5. Iterating

Things that the team can consider in future iterations:

- Consider more details of existing personas in design. Specifications like the Operating System or device that a persona uses can affect the design decisions.
- Add a Teaching Assistant (TA) persona: We did not consider the TA perspective because no TA was present in the videos. But TAs play a big role in teaching, and use the software differently, and hence deserve a persona to ensure that their needs are considered in the design.
- Repeat the observations with in-person classes: Even though we are confident that our findings can be generalized to in-person classes, we strongly suggest observation of in-person classes.
- Santos remarked that a drawback of using projectional editors is that students will not learn to correct syntax errors [43]. We hypothesize that using a projectional editor would save enough time to add a parser assignment to the course giving students a mental model of where the syntax errors come from, which benefits them both as programmers and computer scientists.

Bibliography

[1] R. Anderson. *Work Ethnography and System Design. The Encyclopedia of Micro-Computers 20, A. Kent and JG Williams*. 1997.

[2] M. Arbib. Review of'The Sciences of the Artificial'(Simon, HA; 1969). *IEEE Transactions on Information Theory* 16(6) (1970), 803–804.

[3] B. Basir and R. Salam. Tacit requirements elicitation framework. *ARPN J. Eng. Appl. Sci* 10(2) (2015), 572–578.

[4] B. A. Bottos and C. M. Kintala. Generation of syntax-directed editors with text-oriented features. *The Bell System Technical Journal* 62(10) (1983), 3205–3224.

[5] M. M. Burnett and B. A. Myers. Future of end-user software engineering: beyond the silos. In: *Future of Software Engineering Proceedings*. 2014, 201–211.

[6] W. Burnett. d. school Design Thinking Bootcamp (2014).

[7] E. D. Canedo and R. P. da Costa. The use of design thinking in agile software requirements survey: a case study. In: *International Conference of Design, User Experience, and Usability*. Springer. 2018, 642–657.

[8] E. D. Canedo, A. C. D. S. Pergentino, A. T. S. Calazans, F. V. Almeida, P. H. T. Costa, and F. Lima. Design Thinking Use in Agile Software Projects: Software Developers' Perception. In: *ICEIS (2)*. 2020, 217–224.

[9] A. Cooper. The inmates are running the asylum. In: *Software-Ergonomie'99*. Springer, 1999, 17–17.

[10] A. Cooper. *About Face 2.0: The Essentials of Interaction Design*. Wiley Publishing, 2003. ISBN: 0764526413.

[11] S. De and V. Vijayakumaran. A Brief Study on Enhancing Quality of Enterprise Applications using Design Thinking. *International Journal of Education and Management Engineering* 9(5) (2019), 26.

[12] *Design Thinking: get a quick overview of the history*. https://www.interaction-design.org/literature/article/design-thinking-get-a-quick-overview-of-the-history. Accessed: 2021-11-30.

[13] Donzeau-Gouge. A Structure Oriented Program Editor: a First Step Towards Computer Assisted Programming. In: *Computing Symp., Antibes*. 1975.

[14] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. *Programming environments based on structured editors: The MENTOR experience*. Tech. rep. INSTITUT NATIONAL DE RECHERCHE D'INFORMATIQUE ET D'AUTOMATIQUE ROCQUENCOURT ..., 1980.

[15] V. Donzeau-Gouge, B. Lang, and B. Melese. Practical applications of a syntax directed program manipulation environment. In: *Proceedings of the 7th international conference on software engineering*. 1984, 346–354.

[16] *Ethical Conduct for Research Involving Humans*. https://ethics.gc.ca/eng/documents/tcps2-2018-en-interactive-final.pdf. Accessed: 2021-12-10.

[17] H. Ferreira Martins, A. Carvalho de Oliveira Junior, E. Dias Canedo, R. A. Dias Kosloski, R. Ávila Paldês, and E. Costa Oliveira. Design thinking: Challenges for software requirements elicitation. *Information* 10(12) (2019), 371.

[18] D. B. Garlan and P. L. Miller. Gnome: An introductory programming environment based on a family of structure editors. *ACM Sigplan Notices* 19(5) (1984), 65–72.

[19] K. Goffin, F. Lemke, and U. Koners. *Identifying hidden needs: creating breakthrough products*. Springer, 2010.

[20] W. J. Hansen. User engineering principles for interactive systems. In: *Proceedings of the November 16-18, 1971, fall joint computer conference*. 1972, 523–532.

[21] W. Hansen. Graphic editing of structured text. In: *Advanced Computer Graphics*. Springer, 1971, 681–700.

[22] C. Heath and P. Luff. *Technology in action*. Cambridge university press, 2000.

[23] J. Hehn and F. Uebernickel. The use of design thinking for requirements engineering: an ongoing case study in the field of innovative software-intensive systems. In: *2018 IEEE 26th international requirements engineering conference (RE)*. IEEE. 2018, 400–405.

[24] V. T. Heikkilä, D. Damian, C. Lassenius, and M. Paasivaara. A mapping study on requirements engineering in agile software development. In: *2015 41st Euromicro conference on software engineering and advanced applications*. IEEE. 2015, 199–207.

[25] *how IDEO Uses Customer Insights to Design Innovative Products Users Love*. `https://www.usertesting.com/blog/how-ideo-uses-customer-insights-to-design-innovative-products-users-love`. Accessed: 2021-11-30.

[26] A. Husaria and S. Guerreiro. Requirement Engineering and the Role of Design Thinking. In: *ICEIS (2)*. 2020, 353–359.

[27] M. B. Jensen, F. Lozano, and M. Steinert. The origins of design thinking and the relevance in software innovations. In: *International Conference on Product-Focused Software Process Improvement*. Springer. 2016, 675–678.

[28] D. Kelley and T. Brown. An introduction to Design Thinking. *Institute of Design at Stanford. doi: https://doi. org/10.1027/2151-2604/a000142* (2018).

[29] G. Klein, B. Moon, and R. R. Hoffman. Making sense of sensemaking 1: Alternative perspectives. *IEEE intelligent systems* 21(4) (2006), 70–73.

[30] J. Kolko. Sensemaking and framing: A theoretical reflection on perspective in design synthesis (2010).

[31] K. Kumar, D. Zindani, and J. P. Davim. Introduction to Design Thinking. In: *Design Thinking to Digital Thinking*. Springer, 2020, 3–15.

[32] B. Lang. On the usefulness of syntax directed editors. In: *Advanced Programming Environments*. Springer. 1987, 47–51.

[33] P. Lucena, A. Braz, A. Chicoria, and L. Tizzei. IBM design thinking software development framework. In: *Brazilian Workshop on Agile Methods*. Springer. 2016, 98–109.

[34] M. S. Mirza and S. Datta. Developing Software Using Agile and Design Thinking Framework. In: *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE. 2020, 1819–1823.

[35] N. Murauer. Design thinking: using photo prototyping for a user-centered interface design for pick-by-vision systems. In: *Proceedings of the 11th PErvasive Technologies Related to Assistive Environments Conference*. 2018, 126–132.

[36] A. S. Negi, J. Kumar, S. Luqman, K. Shanker, M. Gupta, and S. Khanuja. Recent advances in plant hepatoprotectives: a chemical and biological profile of some important leads. *Medicinal Research Reviews* 28(5) (2008), 746–772.

[37] M. Neumeier. *The 46 Rules of Genius: An Innovator's Guide to Creativity*. New Riders, 2014.

[38] P. Newman, M. A. Ferrario, W. Simm, S. Forshaw, A. Friday, and J. Whittle. The role of design thinking and physical prototyping in social software engineering. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, 487–496.

[39] D. A. Norman. *The Design of Everyday Things*. 1988.

[40] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In: *Proceedings of the Conference on the Future of Software Engineering.* 2000, 35–46.

[41] J. C. Pereira and R. de FSM Russo. Design thinking integrated in agile software development: A systematic literature review. *Procedia computer science* 138 (2018), 775–782.

[42] J. J. Sanders, E. Caponigro, J. D. Ericson, M. Dubey, J.-N. Duane, S. P. Orr, W. Pirl, J. A. Tulsky, and D. Blanch-Hartigan. Virtual Environments to Study Emotional Responses to Clinical Communication: A scoping review. *Patient Education and Counseling* (2021).

[43] A. L. Santos. Javardise: a structured code editor for programming pedagogy in Java. In: *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming.* 2020, 120–125.

[44] J. H. Schaeffer. *Videotape: New techniques of observation and analysis in anthropology.* De Gruyter Mouton, 2009.

[45] H. A. Simon. *The Sciences of the Artificial (3rd Ed.)* Cambridge, MA, USA: MIT Press, 1996. ISBN: 0262691914.

[46] V. Stray, R. Hoda, M. Paasivaara, and P. Kruchten. *Agile Processes in Software Engineering and Extreme Programming: 21st International Conference on Agile Software Development, XP 2020, Copenhagen, Denmark, June 8–12, 2020, Proceedings.* Springer Nature, 2020.

[47] A. Suzianti and G. Arrafah. User Interface Redesign of Dental Clinic ERP System using Design Thinking: A Case Study. In: *Proceedings of the 2019 5th International Conference on Industrial and Business Engineering.* 2019, 193–197.

[48] A. Suzianti, A. D. Wulandari, A. H. Yusuf, A. Belahakki, and F. Monika. Design Thinking Approach for Mobile Application Design of Disaster Mitigation Management. In: *Proceedings of the 2020 2nd Asia Pacific Information Technology Conference*. 2020, 29–33.

[49] *Teaching Empathy Through Design Thinking*. `https://www.edutopia.org/blog/teaching-empathy-through-design-thinking-rusul-alrubail`. Accessed: 2021-11-30.

[50] T. Teitelbaum and T. Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM* 24(9) (1981), 563–573.

[51] G. Van Dijk. Design ethnography: Taking inspiration from everyday life. *Service Design Thinking* (2010).

[52] R. Vivian, K. Falkner, and C. Szabo. Can everybody learn to code? Computer science community perceptions about learning the fundamentals of programming. In: Nov. 2014.

[53] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In: *International Conference on Software Language Engineering*. Springer. 2014, 41–61.

[54] S. Walker. Design Think. In: *Design Realities*. Routledge, 2018, 63–64.

[55] S. P. Ylirisku and J. Buur. *Designing with Video: Focusing the user-centred design process*. Springer Science & Business Media, 2007.