Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Jørgensen, Joachim William Hegvold
Kvannli, Simon

Efficient Generation of Parsons Problems for Digital Programming Exams in Inspera

Master's thesis in Masters of Informatics
Supervisor: Sindre, Guttorm
June 2019

Jørgensen, Joachim William Hegvold
Kvannli, Simon

Efficient Generation of Parsons Problems for Digital Programming Exams in Inspera

Master's thesis in Masters of Informatics
Supervisor: Sindre, Guttorm
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

For generations, programming exams at the Norwegian University of Science and Technology (NTNU) have been performed using pen and paper. But recently, digital exams were introduced by utilizing the Inspera Assessment platform, which offers a number of new question types. However the creation of some of these question types are not always as efficient or user friendly as the course supervisors would prefer, which is what this thesis proposes a solution for.

The aim of this thesis was to design and create a prototype for efficient generation of drag and drop Parsons problems for digital programming exams in the Inspera Assessment platform. The thesis also evaluates the effect of this prototype, in regards to usability, compared to the manual creation of said questions. The proposed system aims to automate the task generation process, as a mean to increase efficiency, effectiveness, and satisfaction.
The prototype was designed to generate drag and drop tasks using the IMS Question and Test Interoperability specification (QTI) format version 2.1, as required by Inspera to import external questions. QTI 2.1 is a standard format created to facilitate interoperability between systems, meaning that the work of this thesis is not restricted to Inspera alone, but can also be applied to other platforms supporting QTI 2.1.

Based on user testing, the proposed automation system showed significant improvements in all aspects of usability, meaning efficiency, effectiveness, and satisfaction among the test participants. Overall, the system was highly preferred for creating drag and drop Parsons problems instead of manually creating them using Inspera.

Sammendrag

Programmeringseksamener ved Norges teknisk-naturvitenskapelige universitet (NTNU) har tidligere blitt utført med penn og papir, men gjennom et sammarbeid med Inspera Assesment har universitetet nå introdusert digitale eksamener, som tilbyr et mangfold av flere nye oppgavetyper. Bakdelen med noen av disse oppgavetypene er at de kan være tidkrevende og lite brukervennlig for faglærere å lage. Denne masteroppgaven presenterer en mulig løsning for denne problemstillingen.

Målet med denne masteroppgaven var å designe og utvikle en prototype for effektiv generering av dra og slipp Parsons problems for digitale programmeringseksamener i Inspera Assesment plattformen. Denne oppgaven evaluerer også effekten av denne prototypen, med tanke på brukervennlighet, i forhold til den manuelle prosessen med å lage disse oppgavene. Det foreslåtte systemet sikter på å automatisere deler av prosessen for å øke effektivitet og tilfredshet blant faglærere.
Prototypen var designet for å generere dra og slipp-oppgaver ved å bruke IMS Question and Test Interoperability specification (QTI) format versjon 2.1, som er påkrevd av Inspera for å importere eksterne oppgaver. QTI 2.1 er et standard format laget for å legge til rette for interoperabilitet mellom systemer, noe som betyr at resultatet av denne masteroppgaven ikke er begrenset til bare Inspera, men kan også brukes av andre plattformer som støtter QTI 2.1.

Basert på brukertester viste den foreslåtte prototypen signifikante forbedringer i alle aspekter av brukervennlighet, altså effektivitet og tilfredstilhet blant testdeltagerene. Alt i alt var systemet foretrukket framfor å manuelt lage dra og slipp Parsons problems i Inspera.

Preface

This thesis concludes our five year journey at the Norwegian University of Science and Technology (NTNU), at the Department of Computer Science (IDI). The project was conducted during the fall of 2018 and spring of 2019, and is the final submission for the degree of Masters of Informatics.

We would like to offer a big thank you to our supervisor, Professor Guttorm Sindre, for providing valuable academic guidance and feedback throughout the project. We would also like to extend our gratitude to Madeleine Lorås for insightful feedback on the report structure, and to the people who took time out of their busy schedules to perform user tests of the prototype. And finally, we would like to thank our families for their continuous support throughout the years.

Trondheim, June 2019

Joachim Jørgensen and Simon Kvannli

Contents

List of Tables xvi

List of Figures xx

Abbreviations 1

1 Introduction 3

1.1 Motivation . . . 3
1.2 Previous Work . . . 4
1.3 Research Questions . . . 4
1.4 Scope . . . 5
1.5 Contributions . . . 5
1.6 Thesis Outline . . . 6

2 Background 7

2.1 Parson's Problem . . . 7
2.1.1 Good Coding Structure . . . 8
2.1.2 Distractor Options . . . 8
2.1.3 Modified Parsons Problems . . . 9

2.1.4 Efficient Marking 10
2.1.5 Cognitive Load 11
2.2 E-learning and E-assessment 12
2.2.1 QTI 13
2.3 Inspera Assessment 14
2.3.1 Drag and Drop 16
2.3.2 Storage Persistence for Exercises 16
2.4 Drag and Drop Design in Inspera 16
2.4.1 Limitations 16
2.4.2 Drag Area Anchor 18
2.4.3 Design Options 19
2.5 Creating Tasks Manually with Inspera 20
2.5.1 Create a New Task 21
2.6 Automation 24
2.7 Design 25

3 Related Work **31**
3.1 TAO 31
3.2 ONYX Editor 32
3.3 CORT 33
3.4 JS-Parsons 33

4 Research Methodology **35**
4.1 Research Strategies 35
4.1.1 Research Question 1 35
4.1.2 Research Question 2 37
4.2 Data Generation Methods 43

4.2.1 Data Generation Method for the Design and Creation Strategy 43
4.2.2 Data Generation Method for the Experiment Strategy . . . 44
4.3 Evaluation . . . 50
4.3.1 Research Question 1 . . . 50
4.3.2 Research Question 2 . . . 50
4.4 System Development Method . . . 56
4.4.1 Agile Software Development . . . 56
4.4.2 Extreme Programming . . . 58
4.4.3 Software Prototyping . . . 59
4.4.4 Kanban . . . 60
4.4.5 Testing . . . 60

5 Results 63
5.1 Design and Creation . . . 63
5.1.1 Iteration 0 . . . 64
5.1.2 Iteration 1 . . . 67
5.1.3 Iteration 2 . . . 69
5.1.4 Iteration 3 . . . 75
5.1.5 Iteration 4 . . . 79
5.1.6 Iteration 5 . . . 83
5.1.7 Permutations . . . 90
5.1.8 Technology Choices . . . 99
5.1.9 Design . . . 104
5.2 Experiment . . . 107
5.2.1 Quantitative Data . . . 107
5.2.2 Qualitative Data . . . 114

6 Discussion **117**

6.1 Design and Creation Discussion 117

6.1.1 Deciding What to Automate 118

6.1.2 Design, Evaluation Criteria & Priorities 120

6.1.3 Implementation & Testing 122

6.1.4 Other comments 123

6.2 Experiment Discussion 125

6.2.1 Quantitative Discussion 125

6.2.2 Qualitative Discussion 131

7 Conclusion and Future Work **135**

7.1 Conclusion 135

7.2 Future Work 136

Appendix A Models **139**

A.1 Activity Diagram 140

A.2 Development View 141

A.3 Logical View 142

Appendix B User Manual **145**

B.1 Parsons Generator 145

B.1.1 Add New Task 145

B.1.2 Navigate Tasks 145

B.1.3 Edit Task 147

B.1.4 Delete Task 150

B.1.5 Export Task(s) 150

B.1.6 Helper Icons 151

B.2 Uploading to Inspera 152

Appendix C Use Cases **153**

C.1 Iteration 1 153

C.2 Iteration 2 154

C.3 Iteration 3 156

C.4 Iteration 4 158

C.5 Iteration 5 161

Appendix D Requirements **167**

D.1 List of Requirements 167

D.2 Non-functional Requirements 169

D.3 Removed Requirements 170

D.4 List of Improvements Discovered during User Testing 171

Appendix E Transcription Categories **177**

E.1 Inspera 177

E.1.1 Positive Categories 177

E.1.2 Negative Categories 178

E.2 IT artefact 179

E.2.1 Positive Categories 179

E.2.2 Negative Categories 180

Appendix F The Keystroke-Level Model for User Performance Time **181**

F.1 The Keystroke-Level Model using Inspera 181

F.1.1 Adding Drop Areas 181

F.1.2 Adding Drag Areas 183

F.2 The Keystroke-Level Model using IT Artefact 186

F.2.1 Adding a Task 186

Appendix G Declaration of Consent **189**

List of Tables

4.1 Original SUS statements compared to the modified SUS statements used in the experiment. 47

4.2 Interview questions 49

5.6 Web vs Desktop Application 101

5.7 Pros and cons of frameworks 103

5.8 Participant 1 data points 108

5.9 Participant 2 data points 108

5.10 Participant 3 data points 108

5.11 Participant 4 data points 108

5.12 Wilcoxon singed-rank test on time spent 113

5.13 Top 1 positive categories (only one entry in the table) from Inspera, based on frequency. The complete table can be found in Appendix E.1.1 116

5.14 Top 3 negative categories from Inspera, based on frequency. The complete table can be found in Appendix E.1.2 116

5.15 Top 3 positive categories from the IT artefact, based on frequency. The complete table can be found in Appendix E.2.1 116

5.16 Top 3 negative categories from the IT artefact, based on frequency. The complete table can be found in Appendix E.2.2 116

List of Figures

2.1 An unfinished 2D Parsons problem with distractors. Source: js-parsons (Ihantola and Karavirta, 2011; 2019) 10

2.2 A finished 2D Parsons problem with distractors. Source: js-parsons (Ihantola and Karavirta, 2011; 2019) 10

2.3 Simple example of a QTI 2.2 file (IMS Global Learning Consortium) 14

2.4 Types of exercises available in Inspera Assessment 15

2.5 Short demonstration of Inspera 16

2.6 The drag area anchoring effect in Inspera 18

2.7 Initial Parsons problem design 19

2.8 Initial Parsons problem design preview 19

2.9 Initial design problem 19

2.10 Final Parsons problem design in editor 20

2.11 Final Parsons problem design preview 20

2.12 A task impossible to complete 21

2.13 Small algorithm 21

2.14 New drag and drop question 22

2.15 New drag areas in Inspera 23

2.16 New drop areas in Inspera 23

2.17 Manually placed grid 23

2.18 Manually placed grid result 23

3.1 Match Interaction in ONYX Editor 32

3.2 Order Interaction in ONYX Editor 32

3.3 Code Restructuring Tool (CORT) (Garner, 2001) 33

3.4 An example of a 2D Parsons problem using the js-parsons program (Ihantola and Karavirta, 2011) 34

4.1 Task 1 given during experiment 42

4.2 Task 2 given during experiment 42

4.3 Quantitative Model of Usability (Sauro and Kindlund, 2005) . . . 45

4.4 A comparison of mean System Usability Scale (SUS) scores by quartile, adjective ratings, and the acceptability of the overall SUS score (Bangor et al., 2008) 48

5.1 System Architecture v1.0 64

5.2 Sequence Diagram 65

5.3 Mock-up design from Iteration 0 66

5.4 Flowchart of QTI Converter in Iteration 1 68

5.5 Parsing the skeleton XML in JavaScript 68

5.6 Application after Iteration 2 73

5.7 Flowchart of QTI Converter in Iteration 2 74

5.8 System Architecture v2.0 75

5.9 Application after Iteration 3 77

5.10 Flowchart of QTI Converter in Iteration 3 78

5.11 Help popup for ’With indenting’ 81

5.12 Application after Iteration 4 82

5.13 Application after Iteration 5 88

5.14 Flowchart of QTI Converter in Iteration 5 89
5.15 Unit tests for QTI Converter 90
5.16 Run-time unit tests 90
5.17 Code example 91
5.18 Permutation 91
5.19 Permutations represented in Inspera 91
5.20 All correct permutations of the given code example 92
5.21 Permutations represented in Inspera 92
5.22 All false positive permutations of the given code example 93
5.23 DAG indicating precedences among lines of code 94
5.24 Interface for creating a DAG 94
5.25 Transitive closure 95
5.26 Code line vertex with its preceding and following drop area positions. The code line is thus connected to the green drop areas in Inspera. 96
5.27 Final implementation of permutations in the IT artefact 97
5.28 Code example 97
5.29 Java code example 98
5.30 Visualization of time spent with Inspera and the IT artefact 109
5.31 Calculated User Performance Time for both tasks using KLM . . . 110
5.32 The mean of some gathered data points 110
5.33 Task manually created by test participants in Inspera 111
5.34 System Usability Scores gathered from test participants 112
5.35 Mean SUS 112
5.36 Transcription snippet with assigned categories, from one of the interviews 115

6.1 Considerations during the creation of a system for automation . . 118

A.1 The activity diagram for the process of creating a task 140

A.2 Development view of the source code. The highlighted boxes are files added after the user test 141

A.3 The logical view of the source code before the user test (iteration 4) 142

A.4 The logical view of the source code after the user test (iteration 5) 143

B.1 Application Launch on launch 146

B.2 Application with a list of tasks 146

B.3 Application with an empty task (Top of the task page) 146

B.4 Application with an empty task (Bottom of the task page) 146

B.5 Permutations Settings for a task 149

B.6 Permutations w/ False Positives 149

B.7 Permutation warning 149

B.8 Normal Parsons Problem 150

B.9 Parsons 2D Problem 150

B.10 Delete Task 151

B.11 Export Task(s) 151

B.12 Helper Functionality 151

Abbreviations

Abbreviation	Description
QTI	Question & Test Interoperability
GUI	Graphical User Inteface
SUS	System Usability Scale
IT	Information Technology
XML	eXtensible Markup Language
HTML	HyperText Markup Language
XHTML	eXtensible HyperText Markup Language

Chapter 1

Introduction

1.1 Motivation

Since its foundation in 1760, as Det Kongelige Norske Videnskabers Selskab, and through its many fusions to eventually become what is now known as the Norwegian University of Science and Technology (NTNU), written school exams have mostly been performed using pen and paper. But even with the advances in information and communication technologies during the last century, the exam environments themselves have not advanced. That is, until recently, when a partnership with Inspera began, to commence the process of digitalizing the next generation of exams at NTNU.

Digital exams provides a lot of opportunities (Sindre and Vegendla, 2015a), such as automatic grading, mass generation of similar (but unique) questions, problem-oriented or candidate-oriented grading, integration of relevant work tools, and new question types. Advantages can span from saving costs and resources, to providing a more reliable exam environment. By automating the exam process, time spent administrating and grading can be spent elsewhere.
The possibility of generating enormous amounts of non-uniform exam questions can help prevent cheating (Sindre and Vegendla, 2015b), as it to some degree reduces the benefit of glancing at the student on the next desk. When each exam set is unique, cheating becomes more difficult, which in turn creates a more reliable exam environment.
With the variety of question types, as introduced with digital exams, assessments can be designed to best optimise students chances of achieving specific learning outcomes, which better supports constructive alignment (Biggs, 2014).

A specific question type made possible with the introduction of digital exams are Parsons problems (Parsons and Haden, 2006). These are code completion problems, where one is asked to correctly place remaining lines of code in correct order. Parsons problems are proven to be just as effective as code writing tasks, while making it possible to make grading automatic and more efficient (Denny et al., 2008).
There are numerous ways to create Parsons problems in Inspera, where each way has their own advantages and disadvantages. One type of implementation, the one used in this thesis, is by utilizing the drag and drop question type.

The main drawback with this task type is, currently, the amount of time and effort it takes to create them. The drag and drop tasks require a lot of manual work to be shaped into a Parsons problem, and due to this time-consuming process, they are not utilized to their full potential. Creating a more efficient way to generate these tasks for Inspera can make it more practical, profitable, and advantageous to employ and exploit Parsons problem in different situations.
In broader terms, creating more efficient and effective ways to generate tasks for Inspera can make it easier and faster for course supervisors to create exams and assignments, reduce the number of human-made errors, and further support the mass generation of similar tasks. In this thesis, an attempt to improve the creation of drag and drop Parsons problems, in terms of usability, will be made by creating a prototype that automates this process.

1.2 Previous Work

It is important to mention that the development of this IT artefact was a continuation of a project already worked on by one of the researchers. The initial project was a proof of concept / proof of principle demonstrating that the generation of QTI for Inspera was possible and could potentially be used to generate multiple tasks. This master's thesis was thus created to further develop a usable prototype focusing on the generation of drag and drop Parsons problems in Inspera. And while the development of this IT artefact started from scratch, minor parts of the proof of concept were re-used, specifically for parsing and handling XML objects. So while the proof of concept motivated the master's thesis itself, the impact the proof of concept had on the current development, was minimal.

1.3 Research Questions

This section lists the research questions that guided the problem solving process. Two research questions were defined, which both required different approaches in

terms of research strategies:

Research Question 1: How can one design and create a system for efficient generation of Parsons problems for digital programming exams in Inspera?

Research Question 2: What is the effect, in regards to usability, of using the IT artefact to generate Parsons problems for digital programming exams in Inspera, compared to the manual method?

1.4 Scope

With Inspera, the implementation of Parsons problems can be done in numerous ways, with the use of different question types. They can be created using a set of dropdown menus, pairing between code lines and line numbers, copy and paste a given set of code lines into a code editor, or as a drag and drop exercise. Most of these implementations have their own set of drawbacks, e.g. dropdown menus covers following code lines when selected, pairing fails to present the code snippet as a whole, and copy and paste are prone to typing errors. And while there exists multiple ways of creating Parsons problem, a prerequisite for this master's thesis was to utilize the drag and drop question type to generate Parsons problems for Inspera.

The IT artefact developed in conjunction with this thesis is a prototype, and not a fully implemented end product. The reason for this, was that the goal of the IT artefact, and the thesis, was to demonstrate that this domain, generating drag and drop Parsons problems for Inspera, could be automated and made more efficient, effective, and user friendly.

1.5 Contributions

The contributions of this master's thesis is an IT artefact, which can be found at `https://github.com/joachimjorgensen/masteroppgave-js`, that allows for efficient generation of drag and drop Parsons problems in the QTI format. The thesis proves that this process, generating drag and drop Parsons problems, can be automated, which in turn can save time and resources.

The research is not limited to only NTNU and Inspera, as QTI is a standard format for representation of assessment content and results. Thus, the IT artefact has potential to be used in any platform that supports QTI, such as WiseFlow[1], and at any

[1]https://europe.wiseflow.net/

universities that uses Inspera or other QTI-supporting tools.

1.6 Thesis Outline

The thesis is structured as follows:

- Chapter 1 is the introduction of this thesis
- Chapter 2 introduces background theory, as well as an introduction to Inspera
- Chapter 3 describes existing projects done regarding QTI generation and Parsons problems
- Chapter 4 presents the research methodology utilized in this thesis
- Chapter 5 reveals the results of this thesis
- Chapter 6 discusses the revealed results
- Chapter 7 concludes this thesis and suggests future work

Chapter 2

Background

This chapter introduces background theory for topics associated with this thesis, as well as an introduction to Inspera Assessment.

2.1 Parson's Problem

Learning Computer Science can be a challenging and overwhelming process. Many claims have been made about computer science being difficult to learn (Jenkins, 2002), which can result in high dropout and failure rates (Bergin and Reilly, 2005). While multiple studies have been unable to find any alarmingly low pass rates in introductory programming courses (Bennedsen and Caspersen, 2007, Watson and Li, 2014), it is still a well-known fact that learning to program is difficult. B. Du Boulay introduces many of the main difficulties encountered when learning programming, such as understanding the machine itself, mastering syntax and semantics, understanding the standard structures, and mastering the pragmatics of programming (Du Boulay, 1986).

In first year programming courses today, there are a lot of tedious, repetitive exercises that aims to teach syntactic and semantic rules. While this might be regarded as good practice, they are often boring with little to no engagement (Parsons and Haden, 2006). These types of exercises can be referred to as 'drill technique' exercises, and have been part of the curriculum since the beginning of Computer Science. Educators now days are looking for more fun and engaging ways of teaching the rules and best practices of computer programming.

Parsons problems are puzzle tasks where one must choose and place mixed code blocks in the correct order to form the solution (Parsons and Haden, 2006). Instead

of writing code from scratch, Parsons problems reduces the cognitive load required to solve coding tasks by offering completed code blocks, thus reducing some of the complexity of the syntactic and semantic parts of the code (Ericson et al., 2017). This in turn, makes it possible to focus solely on one part of code construction at a time.

Using different types of Parsons problems as a methodology for teaching and testing code understanding amongst students has proven to be just as efficient, but more effective, than fixing and writing code from scratch (Ericson et al., 2017). Parsons problems are a type of completion task, and completion tasks have proven to be a good alternative to code generation tasks in reducing some of the cognitive load while resulting in better learning outcomes (Van Merriënboer, 1990).

2.1.1 Good Coding Structure

Parsons problems usually consists of blocks of well-written code, which help delineate good programming practices. The beginner programmer might not be aware of the best ways to construct a sub routine or function, but their code might still produce the correct results. For example, a student may be unnecessarily repeating parts of a sub routine, which instead could easily be extracted to a single function. The student may never be aware of this unnecessary repetition, due to the generation of correct output. To fight this sub-optimal way of coding, Parsons problems exposes students to good programming practice.

Another side effect of having samples of well-written code is eliminating the process of finding the correct logical or algorithmic approach to the problem (Parsons and Haden, 2006). This might help students who are not as experienced in programming to spend less time on aspects of the task that are not the target of the exercise, and rather focus on what matters. This might help students in the lower end of the brackets to score some points, even though they are not as good at programming.

2.1.2 Distractor Options

Distractor options are redundant and incorrect inputs to a Parsons problem. They are optional, and their sole purpose is to distract students from the correct solution to the task at hand.
Adding distractors to a Parsons problem can have multiple purposes. First, they can be used to target specific aspects of code syntax or common errors, by providing options that highlight these common mistakes. For example, if it is common for students to confuse the functions 'x.length', 'x.length()' and 'len(x)', the in-

correct options could be included as distractors. Adding common errors like this is also why, in most cases, the distractors are used as a tool to increase the difficulty of a given Parsons problem.
Second, the distractors can also be used to make the same tasks look different for different students. By using the same solution together with different distractors, no task will look the same. This could in turn be a valuable tool to reduce cheating during exams. For example, if one has a total of eight distractors in mind, and want to include four of them in every task, the binomial coefficient states that 70 unique tasks can be created:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{8!}{4!(8-4)!} = 70 \tag{2.1}$$

2.1.3 Modified Parsons Problems

There are numerous tricks and tweaks that can be applied to Parsons problems to make them even more interesting, or to target a desirable difficulty for the problem. This is proven to enhance learning outcome (Bjork, 2017) and should be of interest to anyone interested in Parsons problems.

Two-dimensional Parsons problems adds another dimension to the puzzles, where students are required to correctly indent coding blocks as well as placing them in the correct order (Ihantola and Karavirta, 2011). This can be very effective in first year programming courses, especially programming languages like Python, where indentation is essential to a functional code snippet. Correct indentation of code blocks also teaches students the importance of readable code, and good coding practices. Figures 2.1 and 2.2 illustrates the use of a two-dimensional Parsons problem. From this point on, two-dimensional Parsons problems are referred to as *2D Parsons problems*, while *Parsons problem* (or *1D Parsons problem*) refers to the unmodified version.

Another modification one can add to the Parsons problem is having distractors paired with correct code blocks. Paired Parsons Problems exposes students to a binary choice, which can help simplify the task, but also highlight a particular detail of interest. For example, to focus the attention of the task to the importance of semi-colons in Java, one can have the distractor show an identical line of code, except for the semi-colon at the end.

Dynamically adaptive Parsons problems are tasks where the difficulty is dynamically adapted to the persons performance on previous tasks. For example if one

fails on a task, the next problem can be an easier task or some blocks can automatically be locked in to the correct position. These types of adaptive problems have a significant increase in learning outcomes compared to non-adaptive problems, as it keeps the learner in Vygotsky's zone of proximal development (Ericson et al., 2018).

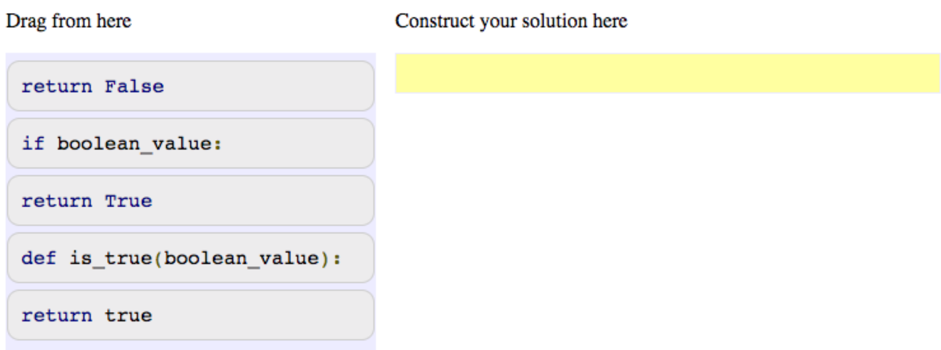

Figure 2.1: An unfinished 2D Parsons problem with distractors. Source: js-parsons (Ihantola and Karavirta, 2011; 2019)

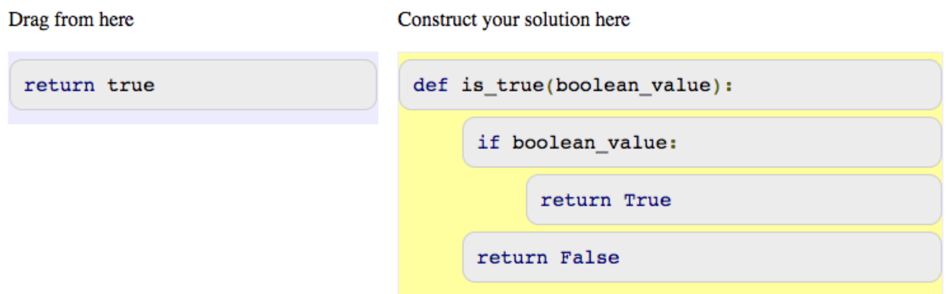

Figure 2.2: A finished 2D Parsons problem with distractors. Source: js-parsons (Ihantola and Karavirta, 2011; 2019)

2.1.4 Efficient Marking

Because Parsons problems are complete well-written code snippets, solving them takes significantly less time than constructing the code from scratch. And the fact that they are pre-constructed blocks of code, the process of marking and grading them are easier and more reliable than traditional code writing tasks (Denny et al., 2008). Parsons problems requires the same set of skills as traditional code writing

exercises, and thus are a good alternative in both exam and regular learning practices.

2.1.5 Cognitive Load

When learning a new topic, the mind has to process the information being presented and save it to the long term memory. During this process, the cognitive load required to handle the information can be split up into three categories (Paas et al., 2003): intrinsic, extraneous, and germane cognitive load.

Let's say the topic of interest is learning to play the piano. The intrinsic cognitive load would be the inherent difficulty of the piece of music you are learning. Learning to play Ludwig van Beethoven's "Für Elise" is substantially easier than learning Franz Liszt's “La Campanella”. This difficulty is immutable, and cannot be eased by any means; it is what it is.

The extraneous cognitive load is generated by the way the new information is presented to the learner. This load can be altered by changing how the information is processed. Back to the the case of learning to play the piano: if you were to learn Beethoven's "Moonlight Sonata", an instructor could tell you all the frequencies of the song, which would increase the extraneous cognitive load, or simply present how the piece should be played, which lessens the extraneous cognitive load.

The germane cognitive load is the process where information is constructed into schemata to be stored in the long term memory (Garner, 2002b). Sweller describes schemas as: *"A schema can be anything that has been learned and is treated as a single entity"* (Sweller et al., 1998), and goes on to describe schema construction having two functions: storage and organization of information in long-term memory, and a reduction of working memory load. Working memory is the part of the mind that actively processes information, i.e. the consciousness.

Every person has a limited amount of resources to use for the cognitive load, which all of the three parts above has to share. As the intrinsic part is said to be immutable, only the latter two can be altered towards our interest. When a task is difficult, i. e. the intrinsic load is heavy, there is less space for the other two. Usually, we want to lessen the extraneous load, and increase the germane load.
Garners says in his article "Reducing the Cognitive Load on Novice Programmers" (Garner, 2002b), that the germane load can be increased by removing parts of the solution, and let the students finish these incomplete examples. This encourages schema creation in long term memory, and thus should help achieve a better learning outcome, which is why Parsons problems are a good alternative to traditional coding exercises.

2.2 E-learning and E-assessment

There are many definitions of e-learning, where most are either too broad and inclusive, or too specific (Piotrowski, 2009). By some authors, e-learning is defined as the use of any electronic media in any learning scenario, but this broad definition would even categorize the use of a laser pointer as e-learning. Tavangarian et al. (Tavangarian et al., 2004) defines e-learning as all forms of electronic supported learning and teaching that has a procedural character to aid individuals learning, which again is a too specific definition for most cases. A more suiting definition might be the one by Clark & Mayer (Clark and Mayer, 2016), which defines e-learning as instructions delivered on a digital device that is intended to support learning. These are devices such as desktop computers, laptops, tablets, smart phones etc.
E-learning systems have the potential to create customized training and learning, tailored content, instructional methods, and spark engagement (Clark and Mayer, 2016). Due to its scalability, organizations can use e-learning to save time and costs when teaching and training students or employees. E-learning are used in more and more types of formal education and is a great tool for both individual learning and instructor-led learning. Piotrowski (Piotrowski, 2009) suggest six activities that characterize and define an e-learning platform:

- **Creation**: The production of learning and teaching materials by instructors
- **Organization**: The arrangement of materials for educational purposes
- **Delivery**: The publication and presentation of the materials
- **Communication**: The computer-mediated communication between students and instructors and among students
- **Collaboration**: Students or instructors jointly working on files or projects
- **Assessment**: The formative and summative evaluation of learning progress and outcomes, including feedback

In educational use, e-learning platforms also allows for continuous monitoring and assessment (*formative assessment*), instead of a single assessment at the end of a course (*summative assessment*), which is preferable for both instructors and students (Piotrowski, 2011). The usually time-consuming activities, such as assessment, can be automated and made more efficient with *computer-aided assessment*,

or *e-assessment*. Multiple-choice tests can, for example, be completely automatically assessed by the platform.

2.2.1 QTI

Creating platform independent tests promote reusability, longevity, and can mitigate costs of creation and ensure sustainability of intellectual assets (Lay, 2004, Piotrowski, 2011). And to create platform independent tests, one needs a standard and platform-neutral file format to create these tests with.

The IMS Question & Test Interoperability (QTI) specification describes a data model for the representation of question and test data, and their corresponding results reports. The main purpose of the QTI specification is to *"define an information model and associated binding that can be used to represent and exchange assessment items"* (IMS Global Learning Consortium). This yields a standard format and allows for assessment materials to be authored and delivered on multiple systems interchangeably. The QTI specification specifies a standard XML data format (XML data binding) to represent assessment content and results. By agreeing to one standard format, different systems and tools can import, export, and exchange the same questions. This international e-learning technical standard defines a set of interaction types which can be used to create different question types, such as multiple-choice, filling in blank, matching, etc.

The QTI format focuses on features (as stated in (IMS Global Learning Consortium)) such as:

- **Interoperate**: Exchanging a wide range of questions and tests. Enabling testing from small quizzes to formal summative high-stakes tests.
- **Accessible**: Supports accessible assessments to accommodate users specific presentation needs and preferences.
- **Open**: Open format gives users full ownership of their test content.
- **Innovative content**: Standard and custom item authoring.
- **Test delivery**: Accessible, adaptive assessments tailored to fit the needs of the user with their range of access devices
- **Repositories**: Item/test banks and the collection of results for analytics.

The QTI format has multiple versions that support varying features and are built on different models, which utilizes completely different XML structures. QTI 1.*x*

are therefore completely incompatible with QTI 2.*x* (Lay, 2004). In this project, the latter version was used, and an example of a QTI 2.2 file can be seen in Figure 2.3.

```
▼<assessmentItem xmlns="http://www.imsglobal.org/xsd/imsqti_v2p2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.imsglobal.org/xsd/imsqti_v2p2 http://www.imsglobal.org/xsd/qti/qtiv2p2/imsqti_v2p2p2.xsd"
 identifier="choice" title="Unattended Luggage" adaptive="false" timeDependent="false">
 ▼<responseDeclaration identifier="RESPONSE" cardinality="single" baseType="identifier">
   ▼<correctResponse>
      <value>ChoiceA</value>
    </correctResponse>
  </responseDeclaration>
 ▼<outcomeDeclaration identifier="SCORE" cardinality="single" baseType="float">
   ▼<defaultValue>
      <value>0</value>
    </defaultValue>
  </outcomeDeclaration>
 ▼<itemBody>
    <p>Look at the text in the picture.</p>
   ▼<p>
      <img src="images/sign.png" alt="NEVER LEAVE LUGGAGE UNATTENDED"/>
    </p>
   ▼<choiceInteraction responseIdentifier="RESPONSE" shuffle="false" maxChoices="1">
      <prompt>What does it say?</prompt>
      <simpleChoice identifier="ChoiceA">You must stay with your luggage at all times.</simpleChoice>
      <simpleChoice identifier="ChoiceB">Do not let someone else look after your luggage.</simpleChoice>
      <simpleChoice identifier="ChoiceC">Remember your luggage when you leave.</simpleChoice>
    </choiceInteraction>
  </itemBody>
  <responseProcessing template="http://www.imsglobal.org/question/qti_v2p2/rptemplates/match_correct"/>
</assessmentItem>
```

Figure 2.3: Simple example of a QTI 2.2 file (IMS Global Learning Consortium)

Piotrowski (Piotrowski, 2011) also discusses some of the main flaws with IMS QTI. These are shortcomings such as the different QTI versions being incompatible and not knowing which parts of the IMS QTI that is supported by a platform (authoring, delivery of tests, assessment, etc.).

2.3 Inspera Assessment

Inspera Assessment (also referred to as just Inspera) is a system for digital assessment of exams, home exercises, projects, bachelor projects, master theses, and oral and practical exams. Inspera Assessment is a digital e-assessment tool that not only supports assessment, but also creation, organization, delivery, communication and collaboration, as mentioned in Section 2.2. While it mainly is an e-assessment tool, Piotrowskis (Piotrowski, 2009) definition would categorize Inspera Assessment as an e-learning platform due to its support of the mentioned features. Nevertheless, Insperas own definition is as follows (Inspera AS, 2019):

> *"Inspera Assessment is a cloud-based assessment platform supporting the entire examination process, including planning, designing, delivering, invigilating, marking & annotating, sharing, and improving."*

Inspera supports a wide variety of question types, such as multiple choice, pairing, and drag-and-drop tasks. Figure 2.4 shows an overview of some of the supported tasks as of the writing of this thesis. Here one can find three categories of questions: *Automatically marked*, *manually marked*, and *not marked*. The automatically marked questions all have the possibility to be automatically assessed by Inspera, and in this category one can find questions such as multiple choice and drag and drop. The manually marked questions has to be manually assessed and consists of questions such as essay, programming, and math working. The final category lets the user add descriptive pages to a test, such as a background history or introduction text, which expects no answer from the participants.

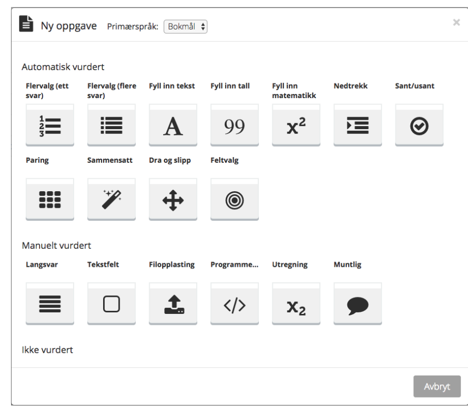

Figure 2.4: Types of exercises available in Inspera Assessment

Inspera uses the IMS QTI 2.1 specification to model all their questions and tests. By using this standard, Inspera can import and export tests from and to other assessment and question bank systems.

Figure 2.5 shows a simple demonstration of how Inspera works. First, one can see the question set, which contains one or more tasks. The question set can be seen as a single test or an exam with multiple tasks, and this question set is what students will be asked to solve. Second, the middle image of Figure 2.5 shows the creation of a single multiple choice question in Insperas own task editor. While the QTI specification supports more tasks and more options than Inspera, only features supported by Inspera will work in their editor. This means that importing a multiple choice question with extra details or features, such as HTML styling, wont necessarily be applied in Inspera unless their editor supports the feature. Once the multiple choice task is created in the editor, it is ready to be used together with the other tasks in the question set. In the last image, one can see the actual task as it will appear for a student solving it. While this demonstration is extremely simplified, it shows the main steps for creating tests in Inspera Assessment.

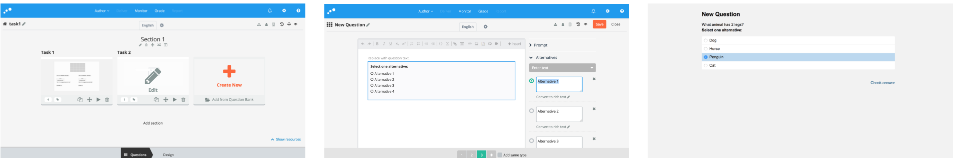

Figure 2.5: Short demonstration of Inspera

2.3.1 Drag and Drop

As mentioned, Inpsera Assessment provides the possibility of creating drag and drop tasks, which is one of the question types that can be automatically assessed. A prerequisite for this research, was to use the drag and drop question type for generating Parsons problems. While some of the other question types could be implemented to create Parsons problems, the drag and drop type in Inspera has the most features similar to those of the js-parsons program (Ihantola and Karavirta, 2011). The design and adaptation of this question type is further described in Section 2.4.

2.3.2 Storage Persistence for Exercises

In Inspera, all exercises are saved and can be reused. This results in a pool of exercises to choose from when creating question sets, making it possible for all students to receive different subsets of exercises.

2.4 Drag and Drop Design in Inspera

In this section, the actual design of the drag and drop tasks as Parsons problems in Inspera will be introduced. The general design of a task in Inspera is concerned with the actual look and behavior of the task. The creation and evaluation of the task design itself is **not** a part of this research, and was already predefined when the research started. This introduction gives a better understanding as to why the tasks look the way they do, and some of the general problems with using Insperas drag and drop tasks to create Parsons problems.

2.4.1 Limitations

To understand some of the reasoning behind the design decisions, one must first address a few features and limitations Inspera has. It is also important to note that the limitations mentioned here are present during the writing of this thesis, and might be up for change in future versions of Inspera.

In Inspera, all question areas have a fixed with of 600 pixels. The reasoning behind this is probably to make sure all screens can in some way support the digital exam. But this 600px width can be a rather cumbersome limitation when working with code, since longer lines of code often occurs. The height of the drag and drop questions can be customized, but the width is fixed. The main problem that occurs because of this, is that the drag areas (elements to be dragged by the user to a destination) can not be moved outside of the question area by a user. This means that, if one has a 600px wide drag area, it will have no horizontal wiggle-room. And as one can see in Section 2.4.2, horizontal movement might be necessary to properly place the anchor over the correct drop area (the destination where drag areas are to be placed).

A solution to longer lines of code could be to reduce the text size, but Inspera currently supports no styling or resizing of text inside a drag area. This text can be changed when generating the QTI, but Inspera will simply undo all additional HTML styling once it is imported and changed in their editor. So when generating tasks for Inspera, any additional HTML, both styling and functionality that is not already supported by Inspera, will be removed once you edit the task. This means that tasks generated can not bring any additional perks or extra features to Inspera that they do not already support.

The text inside each drag area is also centered, which is quite unusual when working with code. And as one can see in Figure 2.8, centered text makes it harder to align the text with the correct indention as in source code editors. One of the solutions to the centered text problem, and the solution applied, was to find the length of each line of code as the drag areas were generated, and properly adjust the width of the drag area to the exact length of the text. This way, the text would take up the entire drag area and the text would be properly aligned at each column (indention level). This can be seen in Figure 2.11.

It is also important to note that Inspera has no support for giving different drag and drop pairs different amounts of rewarded points. With most Parsons problems, it would be desirable to give different lines of code different points to better match the difficulty of placing that line correctly. But with Inspera, correctly placing a function declaration, which usually is at the top left corner and seen as an easy thing to do, is just as much worth as any other piece of the code.

When manually grading a Parsons problem, one could also award a few points to a student if they did some minor mistakes, such as forgetting a single indention, but with Inspera there is no way of giving partial points if such mistakes occurs. The line of code and indention-level must both be correct to receive points. These limitations makes it impossible to create a point system for marking Parsons problems

as recommended by Denny, Luxton-Reilly, and Simon (Denny et al., 2008). They also show that Parsons problems are quicker to grade, and result in more consistent grades between markers than code writing.

2.4.2 Drag Area Anchor

The drag area anchor is the point Inspera uses to register whether a drag area is inside a drop area or not. This means that if the anchor point of a drag area is inside the drop area, they are registered as an answered pair. In Inspera, this point is positioned in the middle of the drag area, as illustrated in Figure 2.6. So if the middle of the drag area (regardless of mouse position) is inside a drop area, they are registered as a pair chosen by the user.

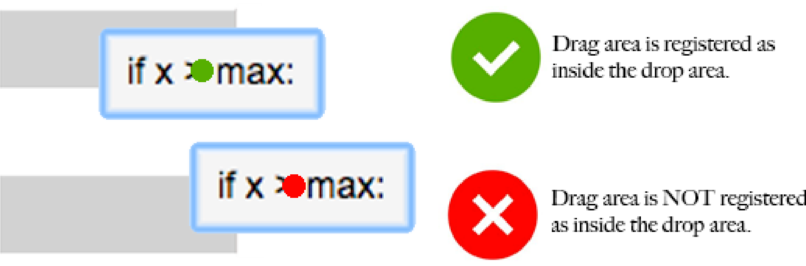

Figure 2.6: The drag area anchoring effect in Inspera

Inspera gives the option to turn off and on "Free placing drag areas". With this option turned off, a drag area will snap to the drop area that the drag area anchor is currently above when released. The snapping simply means that the drag area will be automatically re-positioned to where the left sides of the two areas are aligned. Meanwhile, if the option is turned on, the drag area will not move once it is released by the user.

At this point in time, Inspera has only one type of anchor for the drag and drop questions. There are multiple alternative anchoring rules that might have been better for Parsons problems such as:

- Having the anchor be where the mouse is positioned. If the drag area is to snap to the drop area, it will then move to the drop area which the mouse is hovering when the drag area is released.
- Having the anchor on the far left of the drag area. This might be more

intuitive since code is usually left aligned.

- Remove the anchor point, and pair the areas that overlap the most (And maybe with a small bias towards the middle of the drag area).
- Having the drag area paired up with the drop area closest to the middle anchor.

These alternative anchoring rules might also be assumptions students have when solving the Parsons problems. If it is not clearly explained how the particular anchoring in Inspera works, this could cause a lot of confusion for the users.

2.4.3 Design Options

The initial design of the Parsons problems are shown in Figures 2.7 and 2.8. This design was first created to clearly differentiate between the indentations levels.

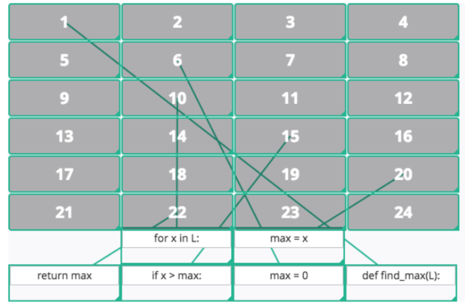

Figure 2.7: Initial Parsons problem design

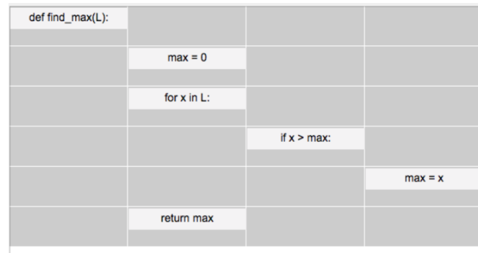

Figure 2.8: Initial Parsons problem design preview

The main problem with this design is that it has poor support of longer lines of code. As mentioned, Inspera has a maximum width of 600px for the drag and drop questions, which in the case of four indent layers create a width of maximum $600/4 = 150$px for each line of code. One could, of course, create drop areas of varying size to better accommodate longer lines of code, but this would give substantial hints for the students as to where the drag area should be placed. Thus, one is forced to make drop areas of the same width at all times, even though the lines of code might vary.

In Figure 2.9, one can see the problem that occurs with longer lines of code. This problem would get even worse if the algorithm had even more indentation levels. An algorithm of, say 8 in-

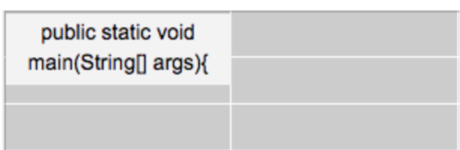

Figure 2.9: Initial design problem

dentation levels, would require each line of code to fit inside a 75px wide block, and with no options to change text size, the block would become a nuisance to read.

Chosen Design

The best solution found, as seen in Figures 2.10 and 2.11, was to create a grid of smaller, fixed drop area sizes. The width of the drag areas on the other hand, was dynamically based on the length of the text inside each drag area. With this solution, the drag areas overflowed the drop areas when placed, which was seen as a benefit as there would be no indication as to which drop area belonged to which drag area, based on the size of the areas. The drawback of this solution, though, was that the anchor points of the drag areas were even more unforgiving. With the smaller drop area sizes, the anchor points in the middle of the drag areas had to be located above the smaller drop areas when dropped, making the position of the drop area somewhat counter-intuitive.

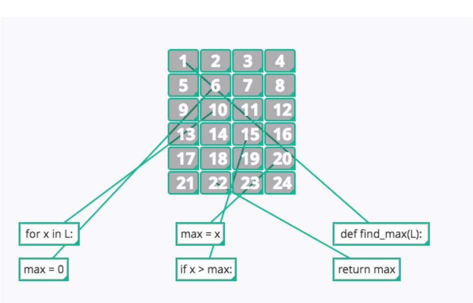

Figure 2.10: Final Parsons problem design in editor

```
def find_max(L):
    max = 0
    for x in L:
        if x > max:
            max = x
    return max
```

Figure 2.11: Final Parsons problem design preview

Another disadvantage with this design, was problems with drop areas close to the edge of the question area. With longer lines of code, and the fact that the anchor point is in the middle of the drag area, drag areas would become too wide to be placed on certain drop areas. And as mentioned, the reason for this is that drag areas can not be moved outside the question area, which would in some cases result in a task impossible to complete. Figure 2.12 shows an example of this problem.

2.5 Creating Tasks Manually with Inspera

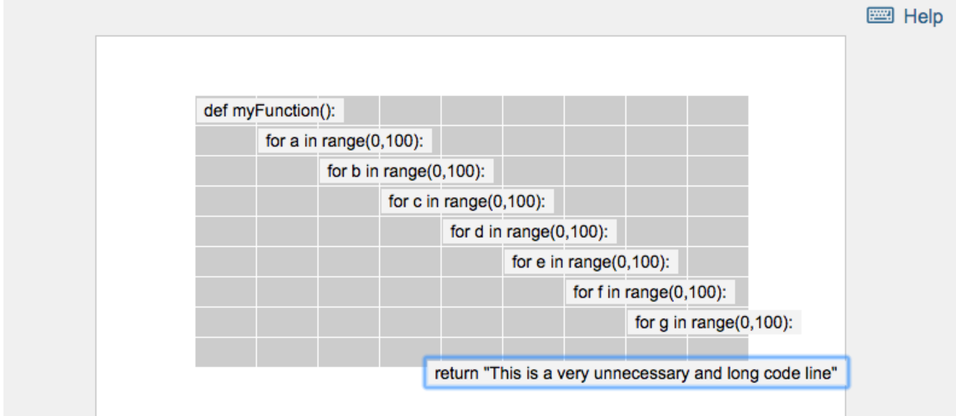

Figure 2.12: A task impossible to complete

This section provides an overview of how drag and drop Parsons problems are manually created with Inspera, which should give a better understanding to where and how the IT artefact will change the current process, and potentially improve efficiency, effectiveness, and overall usability. In addition, this section also highlights some of the shortcomings of Inspera.

```
1 def my_func(some_list):
2   x = 0
3   for(y in some_list):
4     if(y > x):
5       x = y
6   return x
```

Figure 2.13: Small algorithm

Lets now assume there is a short code snippet that should be created as a drag and drop task for the exam, and Figure 2.13 shows this code snippet.

2.5.1 Create a New Task

To begin with, one must create a new question set, which in Inspera is equivalent to the exam document itself. A question set can consist of multiple questions of different types. With an empty question set, the first question can be created.

Inspera provides multiple question types to choose from (see Figure 2.4), and in this case, the drag and drop question type is chosen to create the Parsons problem. The drag and drop type is one of many question types labeled as 'automatic marks' in Inspera, which is one of the most important aspects we want to capitalize on by creating drag and drop questions for the programming exams.

The drag and drop question types in Inspera is in no way tailored for Parsons

problems, but rather created and used for e.g. questions where one has to place a Latin name in the correct position on an anatomy image. This, of course, makes it a challenge to create programming tasks that works like the open-source js-parsons project (Ihantola and Karavirta, 2011).

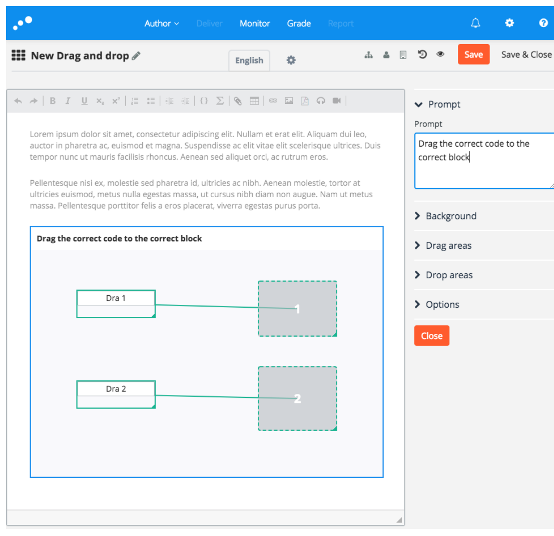

Figure 2.14: New drag and drop question

In Figure 2.14, one can see a newly created drag and drop task with some Lorem Ipsum[1] dummy text added as the task description. A new task initializes with two drag and drop pairs, and requires the user to manually add additional drag and drop areas as seen necessary.

The user first has to add each line of code as drag areas, which in itself is easily done, but as one can see in Figure 2.15, it creates areas that manually has to be re-positioned and changed to fit the text. Inspera offers quite intuitive and clear actions, which makes it easy to understand how to create new drag and drop questions, but it is still a time-consuming and manual process.

In this example, a 2D Parsons problem is to be created with the code from Figure 2.13, which requires the user to create multiple columns of drop areas for each line of code to properly check if the student has understood indentation. The code snippet in this example consists of six lines of code, and has a maximum of four indents, which means the task should have a total of $6 * 4 = 24$ drop areas. In Figure 2.16 one can see some of the drop areas being created, which then needs to be resized and positioned correctly by the user. In Section 2.4.3, a few 2D Parsons problem design options in Inspera are discussed, but this example implements the chosen design for drag and drop tasks from that section.

In Figure 2.17, the drop areas have been correctly placed and connected with the correct drag areas. The biggest challenge regarding the manual placement of the

[1] https://www.lipsum.com/

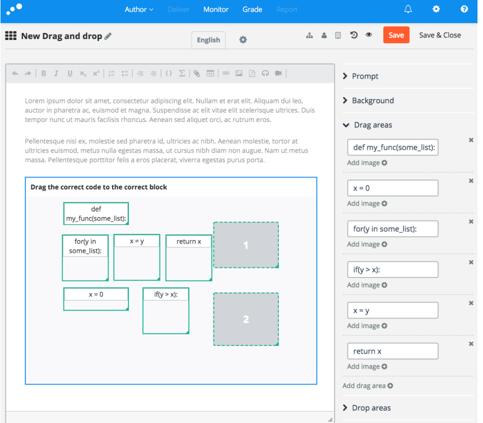

Figure 2.15: New drag areas in Inspera

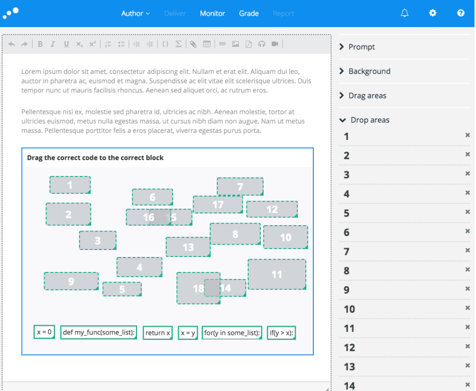

Figure 2.16: New drop areas in Inspera

drop areas is simply to get every area to an equal size with the same distance from each other. The drop areas also have a tendency to snap to a different pixel value once you save the task, resulting in a change in both size and position. The process of resizing and aligning the drop area grid is definitely the most time-consuming aspect of manually creating a drag and drop question in Inspera. Figure 2.18 also shows how uneven the manually created grid becomes. One has to spend a lot of time carefully placing the grid, reviewing how the pixels will change when saving, then reiterating to end up with a somewhat symmetrical grid.

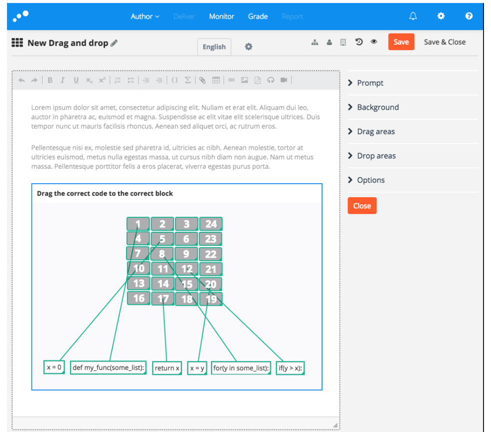

Figure 2.17: Manually placed grid

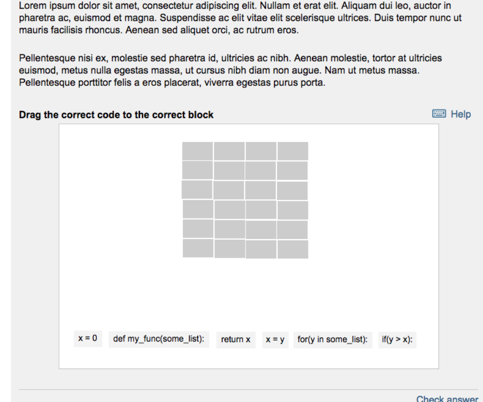

Figure 2.18: Manually placed grid result

Inspera also offers a snap-to-grid functionality that helps the user create more evenly sized drop areas, which has potential to create a more aesthetic grid. The

drawback of this functionality is that the spaces between the drop areas are too large, which increases the probability of misplacing a drag area between two drop areas. At the same time, it is important to mention that having some space between drop areas is a necessity to create the grid effect and make a distinction between drop areas. As a result, the space between the drop areas has to be fine tuned to a size somewhere between zero and the minimum width of the snap-to-grid functionality.

After having created this drag and drop task there are still more features one can customize, such as minimum and maximum points the question can give, but the steps highlighted here were the main steps the artefact aimed to automate.

2.6 Automation

New technology makes it possible to introduce automation into an increasing number of situations and processes. Parasuraman et al. (Parasuraman et al., 2000) discusses the different impacts automation can have in modern systems, and proposes a four-stage model of human-automation interaction with automation, along with distinct evaluation criteria. For this thesis, the most relevant aspects was the automation of information analysis, decision and action selection, and action implementation. Some of the main evaluation criteria for automation is as follows:

- **Mental Workload**: Well-designed automation should reduce human operator mental workload. Although automation does not always reduce mental workload and improve human productivity and efficiency, this should be one of the main criteria.
- **Situation Awareness**: User should have a good 'picture' of what is happening.
- **Complacency**: Users might overly trust the automation. The automation should thus be reliable, or at least provide good ways to monitor the process so failures are detected.
- **Skill degradation**: Automation can make human operators not as skilled in performing the functions that has been automated. Automation must be designed to ensure that reduced situational awareness, complacency and skill degradation does not occur.

It is also difficult to predict how humans will use the automation systems (Parasuraman and Riley, 1997). Discovering the differences between intended and actual

use is important to further improve the automation tool. Automation systems can, for example, lead to over-reliance on automation, underutilization of automation, or implementation without regard for consequences for human performance and operator's authority. Parasurman et al. (Parasuraman and Riley, 1997) also propose multiple strategies to better design and manage automation, such as:

- *Better operator knowledge of how the automation works results in more appropriate use of automation.*
- *Feedback about the automation's states, actions, and intentions must be provided, and it must be salient enought to draw operator attention when he or she is complacent and informative enough to enable the operator to intervene effectively.*
- *The impact of automation failures, such as false alarm rates, on subsequent operator reliance on the automation should be considered as a part of the process of setting automation performance requirements, otherwise, operators may grow to mistrust the automation and stop using it.*
- *The operator's roles should be defined based on the operator's responsibilities and capabilities, rather than as a by-product of how the automation is implemented*
- *Designers of automated systems should consider using alarms that indicate when a dangerous situation is possible ("likelihood" alarms), rather than encouraging the operator to rely on the alarm as the final authority on the existence of a dangerous situation.*

Rovira et al. (Rovira et al., 2007) also find that if automation is performing reliably, complacency is increased, which can lead to more detrimental effects when and if the automation actually fails. This comes from a combination of users simply trusting the automation too much and becoming less trained at fixing or spotting failures. The main point is thus to provide users with good tools for feedback, inspection, and analysis of the automation process if fully reliable decision automation is not guaranteed.

2.7 Design

The aesthetic-usability effect states that designs that are more aesthetic, or prettier, are perceived as more user friendly than less aesthetic designs (Kurosu and

Kashimura, 1995, Lidwell et al., 2010), even if the usability in reality is no different. This is only one of many reasons design needs focus during system development.

There are five common design principles: visibility, feedback, constraints, consistency, and affordance (Preece et al., 2015). These will be described in the following sub chapters, as well as the topics color, confirmation, and user input.

Visibility

When designing a system, visibility refers to the exposure of functions and operations available. Being forced to make an effort to locate desired operations can lead to frustration and dissatisfaction (Preece et al., 2015). For this reason, a software system should make an effort to explicitly display functionality. For example, if it is expected that something should happen when a user clicks on the word 'save', the system has to make this visible. To do that, the word 'save' could be transformed into a button with the label 'save', which makes the save function visible to the user and, with the correct design, easy to find. An example of bad visibility would be to hide the 'save' button in plane sight by not decorating the text whatsoever, making it look like just another fragment of text.

There are several ways to improve visibility, such as reducing the number of hyperlinks to navigate an application (not hiding features on other pages), and highlighting interactive elements. For instance, if everything is located on one page (a single page application), and all actionable elements are transformed to buttons, the application use will most likely be more apparent.

Feedback

If nothing immediately happens when a user clicks on a button, they might become uncertain whether they actually clicked the button or if it is defect. Either way, it stops the users flow through the program, as they have an unexpected stop and get distracted. To combat this, systems should offer feedback whenever an action is made (Preece et al., 2015). Be it either an input field immediately displaying key strokes, a button press activating a loading animation, or an error causing a dialog box.

Constraints

To prevent incorrect user input, a system should utilize constraints to deactivate certain functionalities or operations. For example, when filling out a form, there

might be required fields that disables the submit button until they are filled.

Consistency

Keeping a consistent interface, where there are no surprises, makes the system more user friendly and easier to learn (Preece et al., 2015). Consistency entails mapping the same patterns to the same functionalities, e.g. uniform button design, right clicking the mouse button revealing a context menu, and/or clicking on a company logo redirects the user to the homepage.

Affordance

Affordance is a term conceived by the perceptual psychologist J.J. Gibson (James, 1979), and refers to what the environment can *offer* an actor (human or animal). The ground offers support for standing, a chair offers sitting capabilities, and a light switch offers toggling.
When discussing affordances in the software realm, Norman (Norman, 1999) differentiates between "real" affordance and "perceived" affordance. The former case refers to the affordances of peripherals in the physical world, and should not be confused with the perceived affordances. Keyboards, mice, and monitors, they all might have different affordances, but they are out of the control of a software designer. Instead, a software system can offer perceived affordance, which are the actions the user perceives to be possible. Take the cursor for instance: even though it might change icon from a pointer to a hand when hovering a button on the screen, this change is not what makes a mouse click possible. The mouse is clickable regardless of the position of the cursor, but the user perceives that something is only going to happen when clicking the mouse when hovering a button on the screen. This is a perceived affordance, and not a real one. A real affordance would be, for example, if the mouse had a cover of some sort, preventing a mouse click when not hovering a button on the screen.

Color

The color palette is an important aspect of any design process, as it communicates the ambitions of the software system. Generally, desaturated[2] light colors are perceived as more friendly and professional, while desaturated dark colors more serious and professional. Even though there are no rules regarding color design, there are guidelines worth considering. First, using more than five colors for the

[2]Saturation describes the intensity of a color, and a fully saturated color is considered to be the purest (or truest) version of that color (e.g the base colors red, blue and yellow)

palette might not be advantageous, as humans can only process so many colours at one glance (Lidwell et al., 2010). Second, choosing color combinations that fit well together is favourable, which can be achieved by for example choosing colors adjacent (analogous) or opposing (complementary) on the color wheel. Third, saturated colors are good for grabbing attention. Fourth, cooler colors are more suitable for the background palette, while warmer colors are better for the foreground.

About eight percent of males and .4 percent of females are affected by color vision impairment. The most frequent form of color-blindness is "red-green blindness" which affects the red to green spectrum. The reason for the blindness lies in the cones in the eyes that responds to the light entering. A functional eye has three types of cones, L-, M-, and S-cones, whereas people with color blindness suffers from a complete absence, or dysfunction, of the L- and M-cones (Jenny and Kelso, 2007). When designing software systems, it is important to realize that a significant part of the user base might suffer from this visual impairment, and design with that in mind. Some colors are best to avoid using together, like red and green and purple and blue. This might not always be possible, due to different requirements, conventions, customer preferences etc., which can be met with the addition of visual clues like annotations, shapes, and/or patterns.

Confirmation

Confirmations in software design is a technique to ensure that critical errors are not made. There are two basic types of confirmations: dialog and two-step confirmation. In software design, the former type is mostly utilised, while the latter is more related to hardware design. Dialogs are feedback from the application, waiting for a user response (or confirmation). An example of this is the standard pop-up window, prompting the user to answer ’yes’, ’no’, or ’ok’. When designing such a pop-up dialog, it is important to keep the confirmation message concise, as this is not a dialog that should be cognitive challenging. Overuse of confirmations should be avoided, as repeated exposure lessens its value of importance and they might be ignored (Lidwell et al., 2010).

User Input

Usually, a software system offers interaction, enriching the user experience. Thus, user input is an important part of the interface design. There are no rules when it comes to designing user input, but there are clear guidelines to achieve the best user experience possible. Smith and Mosier (Smith and Mosier, 1986) recom-

mends a set of guidelines to achieve the best design for data entry. While they have hundreds of painfully explicit guidelines, only some are taken directly into consideration in this IT artefact:
First of all, duplicate work done by the user should be avoided. Reason being that if the user is required to enter the same data multiple times, the possibility of erroneous input increases. In addition, users lose focus doing repetitive work. Therefore, users should enter data only once, and then the system should re-use this data whenever necessary. Going even further, associated data entries should be displayed so that users stays a maximum of one click away from all related data entries.
Second, data entries should have well defined areas, so that its functionality is apparent. This entails form fields, check boxes, and buttons.
Lastly, upon completion of data entries, the system should provide explicit feedback, either confirming a successful completion, or displaying an error message.

Chapter 3

Related Work

This chapter describes a few existing projects concerned with generating tasks in the QTI format or generating Parsons problems. There are several studies and research papers regarding Parsons problems, but they typically cover the pedagogical aspects (formulation of questions, learning effect, etc.), and some of these studies are already covered in Section 2.1, but this section focuses specifically on related tools and projects concerned with the creation and generation of tasks.

3.1 TAO

TAO[1] is a test authoring tool developed by Open Assessment Technologies (OAT), which uses the QTI standard for E-test development. It is a commercial-grade open source software, which allows for a wide range of potential customizations.

TAO provides support for most question types in the QTI format, such as choice interaction, slider interaction, text entry interaction, and many more. It also provides some question types in the drag and drop format, e.g. associate interaction, gap match, and order interaction. But even though TAO supports a great deal of question formats, they suffer the same shortcoming as Inspera: the support for Parsons problems as drag and drop exercises is not satisfactory. The common gap match question type allows for drag and drop exercises with distractors, but is unable to create drop areas without a connecting drag area, which makes it unable to generate 2D Parsons problems. The graphic gap match question type, on the other hand, allows for drop areas without a connecting drag area. But this question type requires all drag areas to be images, which means that all code lines would have to

[1] https://www.taotesting.com/

be separate images. In addition, all drop areas must be manually sized, which in most cases would result in an uneven grid.
The remaining drag and drop question types in TAO were less applicable than the two mentioned above, and will therefore not be discussed.

3.2 ONYX Editor

ONYX Editor[2] is an online authoring tool for creating quizzes and interactive assessments with a standard compliant to QTI v2.1. It offers an item (question) bank, and allows for creation of many question types, such as text entry interaction, matrix interaction, choice interaction, just to mention a few. It also supports two types of drag and drop questions: match interaction and order interaction, as seen in Figures 3.1 and 3.2. The first one offers a two column drag and drop question type, where elements from the left column are to be place correctly in the right column. The latter question type offers the possibility to drag elements into correct order, again with only two columns.

This editor provides the possibility to create numerous question types compatible with the QTI format, but the support for creating drag and drop Parsons problems is very limited. For one, code lines has to be manually typed into each drag area, and second, as there are only two columns (one reserved for drag areas and one for drop areas), there is no way of creating 2D Parsons problems.

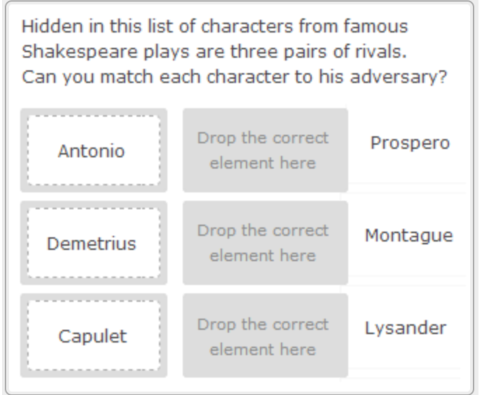

Figure 3.1: Match Interaction in ONYX Editor

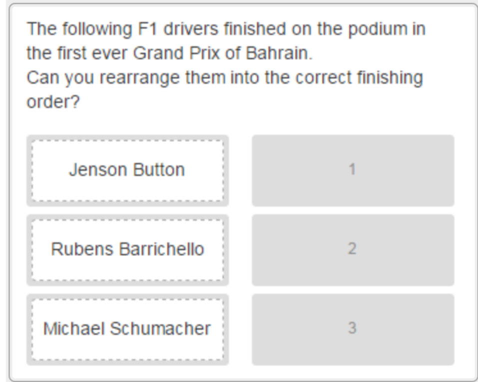

Figure 3.2: Order Interaction in ONYX Editor

[2]https://www.onyx-editor.org

3.3 CORT

Prior to his studies regarding the reduction of cognitive load on novice programmers, Garner (Garner, 2002b) created the Code Restructuring Tool (CORT) (Garner, 2001) for completion of part-complete programming solutions (later known as Parsons problems). The tool was utilized in introductory programming courses at Edith Cowan University, Australia (Garner, 2002a).

CORT is split up into two windows, as seen in Figure 3.3. The left hand window containing code lines and distractors to possibly be moved over to the right hand window, and the right hand window containing the part-complete solution.

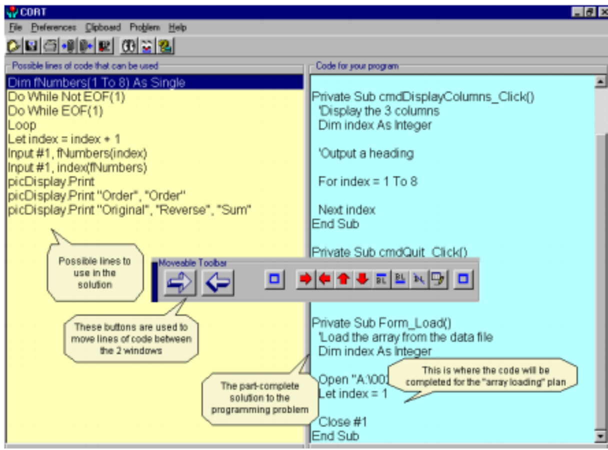

Figure 3.3: Code Restructuring Tool (CORT) (Garner, 2001)

3.4 JS-Parsons

Js-parsons[3], created by Ihantola and Karavirta (Ihantola and Karavirta, 2011), is a JavaScript library for creating and solving Parsons problems. This tool supports features such as 2D Parsons problems, adding distractors, and variables inside statements. In addition, js-parsons supports two modes, one for rearrangements of code lines (only), and another for drag and drop from one area to another (see Figure 3.4).

[3]https://js-parsons.github.io/

Construct a function which returns the largest of the three given arguments. You can change the value of a toggleable element (??) by clicking.

Drag from here

```
return ??
return ??
else:
return ??
```

Construct your solution here

```
def find_largest(a, b, c):
    if b > a:
        if b > c:
            return b
        else:
    elif b > c:
```

New instance Get feedback

Figure 3.4: An example of a 2D Parsons problem using the js-parsons program (Ihantola and Karavirta, 2011)

Chapter 4

Research Methodology

This chapter covers the research methodology chosen for this project. This includes research strategies, data generation methods, data evaluation and analysis methods, and system development methodology. Section 4.1 covers the main research strategies chosen for the two research questions and why. Section 4.2 covers the data generation methods used for the research strategies. Section 4.3 describes how the gathered data was to be evaluated and analyzed, and section 4.4 discusses the chosen system development methodology.

4.1 Research Strategies

This section covers the research strategies chosen for both research questions and describes the reasoning behind these choices. Oates (Oates, 2005) defines six research strategies as overall approaches to answer research questions.

4.1.1 Research Question 1

The first research question was concerned with developing and creating a new software application for efficient generation of Parsons problems, and therefore it would be a natural choice to use the design and creation research strategy. This strategy focuses on the development of new IT products, or IT artefacts as defined by March & Smith (March and Smith, 1995). There are several types of IT artefacts, including: constructs, models, methods, and instantiations. The IT artefact developed in conjunction with this thesis falls mainly into the category of the latter type, instantiations, which is defined as the realization of information systems in its environment as well as the tools used to design the system. In addition, this

thesis provides models such as architectural views, activity diagram, and sequence diagrams to describe the design and creation of the IT artefact, which can be found in Appendix A. The system development methodology chosen for the design and creation strategy is described in Section 4.4

When using the design and creation strategy, it is important to separate between *regular design and creation* versus *design and creation research* (Oates, 2005). The former is the standard industry-based design and creation process, where less changes, less knowledge gained, and less backtracking is seen as a success. The latter, the one used in this research, considers risk taking, backtracking, and knowledge gained as a positive outcome. For this reason, the design and creation was aimed at exploring ideas and features for the proposed IT artefact, and contribute to knowledge by automating a new domain. Research Question 2, described in Section 4.1.2, was used to see if this automation improved the usability or not.

With the design and creation strategy, Oates (Oates, 2005) separates between evaluating IT artefacts where the main objective is to show *'proof of concept'*, *'proof by demonstration'* or *'real-world evaluation'*. Since this research aims at testing the IT artefact on the actual end-users, rather than students or other users, the evaluation of the IT artefact was expected to undergo *'real-world evaluation'*. The reality was that the evaluation ended up somewhere between *'real-world evaluation'* and *'proof by demonstration'*. The testing itself was completed in a somewhat restricted context where all the participants were given the same specific tasks to create, instead of real-world use where they would be able to create tasks for their own needs. This was to simulate its use in real-life context while still gathering good empirical data. The reason behind the simulation and this context-restriction was also that the actual real-life use of the IT artefact were incredibly specific and highly dependent on when exams were created by the people in charge of these exams. To properly complete empirical evaluation in a real-life context, the researchers would have to find users in charge of creating exams or other Inspera exercises, figure out if Parsons problems would be relevant for their needs in this particular case, and figure out exactly when they would be interested in creating these Parsons problems, all while making sure the IT artefact worked perfectly. This approach would also lack data one could compare the new process with and it would be hard to pull off when the development method was expected to follow an iterative process with the testing of an incomplete IT artefact. Due to these complications, the researchers chose to restrict the context of use and the tasks enough to gather comparable data while letting the real end-users get a proper test of how the IT artefact was to be used. This combination of using real end-users while still performing the testing in a somewhat restricted context made the evaluation something in between the two mentioned categories. The consequences of

having real-world evaluation of the IT artefact was that one then needs a separate research strategy dedicated for evaluation (Oates, 2005). This is to properly recognize the need for empirical evaluation in a real-life context, even though this might be somewhat simulated in this research. Since Research Question 1 mainly focuses on the development of the IT artefact, all the evaluation is left to Research Question 2. While a distinction is made between these two research questions, it is important to remember that they complement each other, because while Research Question 1 focuses more on design decisions, initial requirements, and the actual creation of the IT artefact, Research Question 2 focuses on the usability evaluation of the IT artefact from the end-users perspective.

4.1.2 Research Question 2

Alongside the design and creation strategy, where the IT artefact was the main contribution to knowledge, the experiment research strategy was utilized for the second research question. This was to investigate how the new IT artefact was used and perceived in a real-life context by the users. The experiment strategy was a natural follow-up to better understand and evaluate the real-world use of the system.

The experiment strategy was chosen to make the research focus on measuring outcomes and changes made by the manipulation of the independent variable. A case study has too much focus on insight in the IT artefacts' real-life context and does not test hypotheses. This would potentially work if the research only focused on detailed analysis of the IT artefact in use, but since exploring a single instance is not substantial enough to prove if a new tool can improve the current process, a case study was discarded. A survey strategy could potentially be a good choice to get a lot of data in a short time and at a low cost, but due to its focus on breadth of coverage instead of depth, the experiment strategy was seen as a better option. It would also be hard to get useful results out of a survey if the respondents did not fully understand the questions asked. A survey asking lots of people if the created prototype is useful would be difficult since it requires familiarity with Parsons problems, Inspera, and exam creation.

The action research strategy was highly considered instead of experiment strategy due to its concentration on changing and improving a practical issue and its iterative cycle of plan-act-reflect. These iteration cycles would fit well with the iterative improvement of the IT artefact and the focus on creating and refining a system or problem-solving method would work well with this thesis. Action research is also better for exploratory research, especially if there are no precise research questions created. Even though action research could have been the designated

strategy for this research, the experiment strategy was chosen to clearly separate between creation and evaluation and focus more on the causal relationship between the variables.

Hypothesis: *The IT artefact improves usability for generating drag and drop Parsons problems for Inspera, compared to the current manual process.*

The experiment process intended to first observe and measure the manual creation of Parsons problems in Inspera, introduce the IT artefact, then re-observe and re-measure the new process. The independent variable in the experiment was therefore the creation process of a Parsons problem for Inspera. The researchers introduced a new generation process with the IT artefact, and the Parsons problems to create was the same for both processes, making this the controlled variable. The dependent variables are the variables being tested and measured to see the effect of the process change. The dependent variables are listed here as the observations and measurements of the experiment.

The observations and measurements for this research project will be described in more detail in Section 4.2, but mainly included:

- **Project data:** Time to completion
- **Self-report responses:** Subjective assessment of usability with the aspects of effectiveness (successfully meeting objectives), efficiency (amount of effort and resources used to meet objectives) and satisfaction (personal experience).
- **Behavioural counts:** Number of times test participants asked for or needed help, number of system errors, number of user errors made, and number of inaccuracies in the created solutions.

The pre-test consisted of the participant trying to manually create two Parsons problems in Inspera. The entire process was recorded by video to help the researchers accurately measure the data after the observation, and to have the raw data available for later use. Parts of the observation schedule was filled out by the researchers during the experiment, while more detailed measurements and tracking was done afterwards by using the video tape. It was important to hide the notes from the participant during the experiment to avoid having them alter behavior once they saw that, for example, every question was counted. After the pre-test, a questionnaire was conducted to map the perceived usability from the participant. Next, the independent variable was introduced and the participant would have to create the same Parsons problems with the new process. This was the post-test, and

once it was done, the same questionnaire was given. After the pre- and post-tests, an interview was conducted to gather and discover additional ideas, feedback, or requirements. More information about the observation process, interviews and measurements can be found in Section 4.2.2.

All test participants were handed two declarations of consent that disclosed the implications of being a part of this experiment and how the data would be stored and processes. The consent forms signed by all participants can be found in Appendix G.

During the experiment, the participants were given two tasks that they had to create both with Inspera and with the IT artefact. These two programming tasks were already created by the researchers. The idea behind the already created tasks was to simulate that the course-supervisor already had created two decent tasks and to be able to give every participant the same tasks. The actual process behind creating an exam task is much more comprehensive, and the way of working might vary both individually between course supervisors, and also depending on the nature of the task. For example, one way of working might be to (i) decide what to ask about on a high level (e.g., which knowledge and skills to test, what difficulty, etc.), then (ii) do some coding and test a model solution to see if it has the intended level of difficulty, (iii) implementing the task in Inspera, (iv) polishing the question text to make it precise and understandable, then (v) have another teacher quality assure the task. The important thing to note here, is that IT artefact being created here is only trying to support step (iii), while the other tasks are outside the scope of this thesis. This also makes it natural to give the test persons pre-defined tasks where the correctly running code is already made, so only step (iii) is in focus during the experiment.
These pre-defined tasks were both programming tasks written in Python. The size of these tasks had to be somewhat fair, since its obvious that longer lines of code would be much worse for the course-supervisors to manually create in Inspera than shorter ones. If they were to create a 15 line code segment that had a maximum of 5 indentation levels, they would have to manually create, resize, and position 75 drop areas in Inspera. So to avoid fatiguing the participants, the given tasks had to be large enough to give good measurements, but still small enough to complete within a reasonable time-frame. The two tasks were sent to the participants computer as two separate python files to simulate how they would be created by the course-supervisors in real-life.

Since this experiment mainly compares two processes for creating the same tasks, it is important to try to only measure the aspects of the tests that are different for each process. Since the test subjects naturally gets better with practice, overlapping actions or tasks are better to avoid if possible, since they might skew the results.

To better see the results of the independent variable, it is important to remove as many other factors as possible that might influence the results. Here is a list of the main factors and conditions changed or eliminated to improve the internal validity of the experiment:

- The creation of a task description for Norwegian *Bokmål*, Norwegian *Nynorsk*, and English would be required in a real-world case, but since the process was the exact same with and without the independent variable during the experiment, this was eliminated from the experiment.

- The main time measurements were tracked from when the actual creation of the drag and drop task was started until it was saved and closed or previewed. The main idea was to not include the extra time it took to navigate around in Inspera, since this could potentially take a while if the participant had no previous experience with it, and since it was irrelevant for the process this research proposes to change.

- The participants were recommended to perform the experiment on their own computers. This was to make them more comfortable, and to avoid any additional time or frustrations caused by an unfamiliar environment. This would also test if the IT artefact worked as intended on different computers.

- To single out the process of actually implementing the tasks in Inspera, the participants were given images of the exact Parsons problems they were supposed to create. This removed any additional time or stress from trying to figure out how the Parsons problems should be designed. This also meant that the participants did not have to come up with their own code. Figures 4.1 and 4.2 shows the two tasks given during the experiment.

- A single test subject got to perform both the pre-test and the post-test. An alternative could be to do a static group comparison, where half the participants gets to do the manual process, while the other half gets to try the IT artefact. This could eliminate the concern that the participants might get better at Inspera or at the IT artefact in between the different tests and experiment iterations. But for this experiment, the overlap in knowledge from the pre- and post-tests were minimal, making it acceptable to have each par-

ticipant perform both tests.

- The pre-test was, as previously mentioned, the participants making two drag and drop tasks manually, but to improve the validity of the measurements, one can randomly choose whether the IT artefact or the manual process should be the pre-test and which of the given tasks are to be performed first. In other words, the order of the given assignments during the experiment were rearranged to improve validity. If the two tests have any dependent factors influencing the measurements, such as getting better at how Inspera works, this shuffling can reduce or eliminate the error in the measurements.

Factors and conditions to improve the external validity of the experiment:

- Using the real end-users of the IT artefact as test subjects made sure that the participants were representative of the wider population of end-users. The most important factor to improve the statistical significance and generalizability of the research was to increase the number of participants used, but this is discussed further in Section 4.2.

- Perform the experiment one-on-one at the participants own office, making the environment as comfortable and familiar as possible. This natural setting still gives good control over the different variables in the experiment. One could argue that the experiment itself then becomes a quasi-experiment, but since the experiment itself does not take place over a longer period of time in the real-world situation, the experiment still becomes a true experiment where instead of a laboratory, it takes place in the participants own offices. This was also a true experiment due to it being performed in a controlled environment, activities being controlled and the random elements added for internal validity.

Figures 4.1 and 4.2 shows the two tasks every participant were given during the experiment. As previously mentioned, the order of these tasks were randomized and mixed during testing.

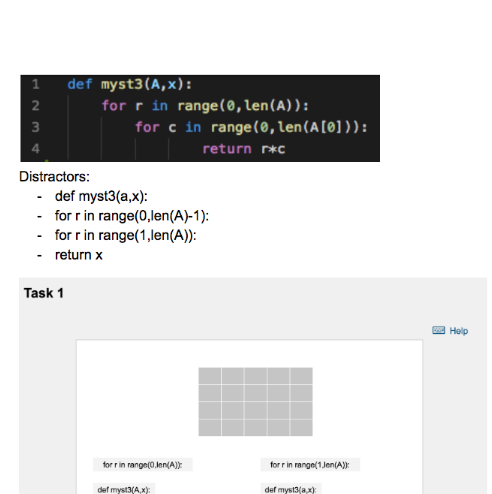

Figure 4.1: Task 1 given during experiment

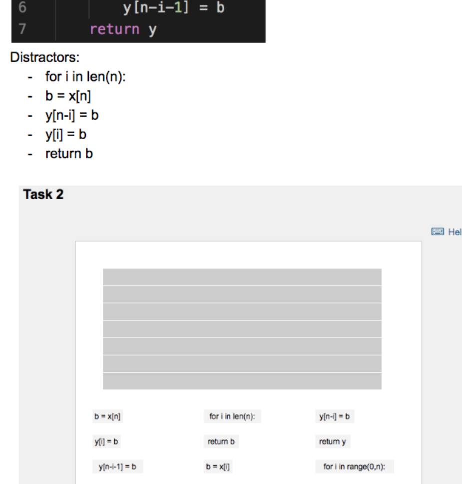

Figure 4.2: Task 2 given during experiment

4.2 Data Generation Methods

4.2.1 Data Generation Method for the Design and Creation Strategy

For the design and creation strategy, the data generation method used was interviews. The main goal of the interviews was to gather end-user requirements, specifications, and design briefs. Since evaluation of the IT artefact was a separate research strategy, the only focus of these interviews was to gather this initial information for the creation of the IT artefact. Gathering and discussing requirements is a complex process, so to delve deeper into specific subjects, a semi-structured interview type was chosen (Oates, 2005). Other data generation methods such as questionnaires could possibly be considered instead of interviews, but the depth and details they generate was unsatisfactory. One-on-one interviews was conducted, instead of group-interviews, to get more honest ideas and opinions, while being free to ask follow-up questions.

When conducting the interview, one researcher took notes while the other focused on the conversation. Audio tape recordings were considered since it is better for 'discovery' and reduces misunderstandings, but due to its time-consuming nature, live transcribing was enough to extract the main requirements from the interviews. To check if the notes and understandings were correct, a summary of the notes was read back to the participant at the end of the interview. This way the interviewee could add further details, change, or clear up any misunderstandings.

The main shortcoming of the data gathering was that the initial list of requirements was based on two interviews with course supervisors at NTNU in charge of creating digital programming tasks and exams with Inspera. This might seem like an extremely low number of people, and it is not enough to make larger generalizations of what all course supervisors want from the IT artefact. The main reason no further people was interviewed for the initial requirements was that a long and detailed list of ranked requirements was already in place after these interviews. Another reason was that, although Parsons problems have received attention internationally since its initial release, it was quite new at NTNU, at least in an exam context following the transition to digital exams. This meant that maybe few teachers had even considered using this problem type yet. The two people interviewed had a lot of domain knowledge, ideas, opinions, and love for the field, making the information gathered here more than enough to get started. One could question the validity of these requirements, but since an agile development methodology was deployed, there was little benefit to spending too much time gathering initial requirements. The main idea was to conduct more interviews regarding the prototype using the experiment strategy, making these interviews somewhat different, as discussed in Section 4.2.2.

The interviews themselves were performed in the course supervisors own offices in the hopes they would be as comfortable as possible and for their own convenience. Another shortcoming of the semi-structured interview was that the initial questions that was asked during the interview could themselves have been biased or leading. An example is the question: "Would automatically generated distractors suggestions be desirable?", which inserts ideas the interviewee wouldn't necessarily come up with themselves. The common answer to such leading questions was "I am not sure, but I guess that would be a nice feature.", showing that questions like this in no way leads to proper discovery of that persons real requirements. And it is important not to lead the witness.

The gathered notes was then structured into a table of functional and non-functional requirements, and all requirements gathered from both these initial interviews and later feedback can be found in Appendix D. These tables also provide information on when the specific requirements were added.

4.2.2 Data Generation Method for the Experiment Strategy

Since the main goal of this research was to compare the usability of the new IT artefact compared to the current process of generating Parsons problems, general usability evaluations and metrics were deployed. Usability with its dimensions is defined as (ANSI, 2001):

> *"The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use."*

Effectiveness metrics relate to percent of task completion, number of errors, number of questions asked, and assistance needed. Efficiency is measured by the mean time to completion and resources expended. And satisfaction focuses on the users subjective opinions of the product with measurements such as satisfaction, usefulness, and ease of use (ANSI, 2001).

Representing these dimensions as a single value for usability is a difficult matter and there have been many attempts to create such a measure (Sauro and Kindlund, 2005). Studies have also shown that the three aspects of usability have a weak correlation and recommends measuring and considering each aspect independently (Frøkjær et al., 2000). So to avoid making additional claims or risky assumptions about the usability, these aspects was considered separately and independently in this research.

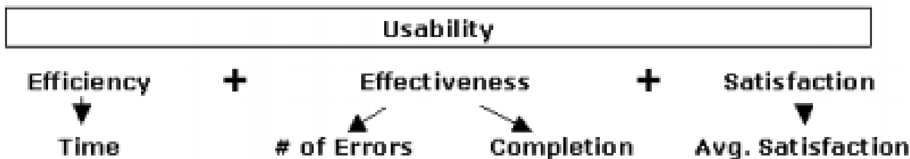

Figure 4.3: Quantitative Model of Usability (Sauro and Kindlund, 2005)

Effectiveness, was measured by looking at the quality of the solution (inaccuracies in the solution), the number of errors made during the process and the number of questions asked / assistance needed.

Efficiency, was measured by task completion time.

Satisfaction, was measured using the System Usability Scale (SUS) questionnaire and a short interview.

The usability and its dimensions was gathered using the following data generation methods

Observation

Note that observations are in effect a survey (Oates, 2005). The survey strategy highlights some additional concerns that researchers using the observation data generation method must take into account. First of all, it is important to clearly define a sampling frame and sampling technique. The sampling frame of relevant people to observe using the IT artefact was course supervisors or professors at NTNU previously or currently in charge of creating digital programming tasks and exams with Inspera. The main idea behind this narrow sampling frame was to ensure real-world evaluation of the prototype, making the real end-users the best candidates for testing. One could argue that anyone with programming expertise might be relevant for testing the IT artefact, but this requires extensive introductions to Inspera before one could properly compare the two procedures. The course supervisors also have more domain knowledge within the field of creating tasks and exams, and can give feedback most people would not even consider. The requirements concerning the IT artefact was simply too specific to let other individuals than course supervisors decide what should and should not be implemented, improved, or changed.

The technique used to select course supervisors from the sampling frame was the non-probabilistic Snowball sampling technique. While the Snowball sampling technique has some deficiencies, such as problems with sampling principles and

engaging with respondents as informal research assistants, the technique is still considered economical, efficient, and good for accessing hidden or unknown populations (Atkinson and Flint, 2001). The supervisor of this master thesis was able to suggest relevant people to the research who would be considered as an end-user of the IT artefact. The sampling frame was the main criteria used when asking for test subjects. The main benefit of this sampling technique was that the researchers could be led directly to the relevant test subjects by taking advantage of the social networks of the identified respondents.

The sampling size was set to around five users, since more elaborate usability tests are often seen as a bad investment of resources in regards to usability problems found. Nielsen & Lauder finds that almost 80% of all usability problems can be found with only five test users (Nielsen and Landauer, 1993).

All of the observations were overt, and while this might trigger the Hawthorne Effect (Adair, 1984), it lets the researchers ask more questions and together with the end-users find the solutions to potential problems or misunderstandings.

The observation also consisted of the *Verbal protocol analysis* or *"think aloud" method*. This method consists of asking the participant to talk and think aloud during the test (Oates, 2005). *"The process of verbalization reveals the assumptions, inferences, misconceptions and problems that the users face while solving problems or performing tasks"* (Benbunan-Fich, 2001). By gathering deeper information as the user is performing the tasks, one avoids the problems where people forget their experiences or justify their actions in retrospective data gathering methods. The subjects should thus make comments while performing the given tasks, which makes it easier for the researchers to understand the users cognitive model, expectations, and misinterpretations of the system (Benbunan-Fich, 2001). Compared to other process tracing techniques, the verbal protocol analysis is likely the most informative and the richest technique (Todd and Benbasat, 1987).

Questionnaires

During the observation process, to map efficiency, effectiveness, and satisfaction of the IT artefact, two questionnaires inspired by the system usability scale (SUS) were conducted. The SUS is a survey scale developed by Brook (Brooke et al., 1996) to give a global view of subjective assessments of a system. The SUS is a mature, robust, and well-established tool that, in this case, is effective for comparing competing implementations of a system and to compare different interfaces (Bangor et al., 2008). It is important to note that the SUS used in this research was slightly modified and adapted to clarify which aspect of the systems and process

that was relevant for the questions, as shown in Table 4.1. The first question in both the pre-test and post-test was extra detailed, while the other questions were more similar to the original SUS statements. This was to let the first question set the frame of mind so the participant knew what aspects of the system that was in question.

Calculating the SUS score is done by following these rules:
"For items 1,3,5,7 and 9 the score contribution is the scale position minus 1. For items 2,4,6,8 and 10, the contribution is 5 minus the scale position. Multiply the sum of the scores by 2.5 to obtain the overall value of SU. SUS scores have a range of 0 to 100." (Brooke et al., 1996)

Original SUS Statements	Pre-test Modified SUS Statements	Post-test Modified SUS Statements
I think that I would like to use this system frequently	I think that I would like to use Inspera frequently to manually create drag-and-drop tasks	I think that I would like to use this system frequently to generate drag-and-drop tasks
I found the system unnecessarily complex	I found that Inspera was unnecessarily complex	I found the system unnecessarily complex
I thought the system was easy to use	I thought Inspera was easy to use	I thought the system was easy to use
I think that i would need the support of a technical person to be able to use this system	I think that i would need the support of a technical person to be able to use Inspera	I think that i would need the support of a technical person to be able to use this system
I found the various functions in this system were well integrated	I found the various functions in Inspera were well integrated	I found the various functions in this system were well integrated
I thought there was too much inconsistency in this system	I thought there was too much inconsistency in Inspera	I thought there was too much inconsistency in this system
I would imagine that most people would learn to use this system very quickly	I would imagine that most people would learn to use Inspera very quickly	I would imagine that most people would learn to use this system very quickly
I found the system very cumbersome to use	I found Inspera very cumbersome to use	I found the system very cumbersome to use
I felt very confident using the system	I felt very confident using Inspera	I felt very confident using the system
I needed to learn a lot of things before I could get going with this system	I needed to learn a lot of things before I could get going with Inspera	I needed to learn a lot of things before I could get going with this system

Table 4.1: Original SUS statements compared to the modified SUS statements used in the experiment.

Figure 4.4 shows a general overview of what passes as an acceptable SUS score for a given system (Bangor et al., 2008), but it is important to note that only evaluating the SUS score in isolation gives little information about the "goodness" of the IT

artefact or Inspera. The main source of data therefore comes from comparing the pre- and post-test SUS scores.

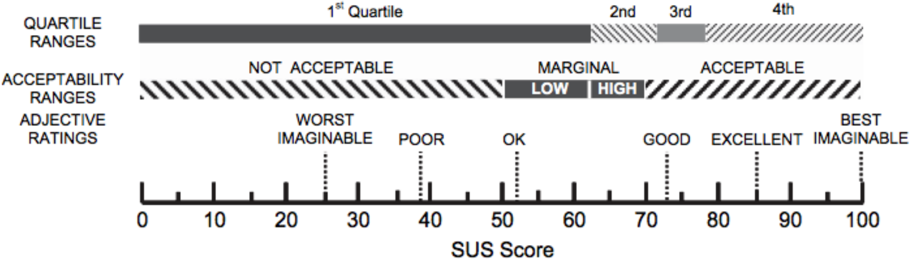

Figure 4.4: A comparison of mean System Usability Scale (SUS) scores by quartile, adjective ratings, and the acceptability of the overall SUS score (Bangor et al., 2008)

As mentioned, the questionnaire was given to the participant during the observation process as a follow up to the pre- and post-test, making it closer to a self-administered questionnaire than a researcher-administrated one. The researchers did not ask the questions themselves, but simply handed the participant a sheet to fill out while they prepared the next part of the experiment. By handing the participant a piece of paper, the researchers avoids having the tone of voice or body language effect the answers. The questionnaire is used to gather standardized data and consists only of closed opinion typed questions. Since the questionnaire works as a supplement to the interview, all open questions are left to the interview section. The response format of the questions follow the 'Likert scale' where one fills out to what degree one agrees or disagrees with the given statement (Brooke et al., 1996). Since the SUS itself is a tried and tested set of questions, the researchers did not have to go through any additional pre-tests or piloting of the questionnaire.

Interviews

As a follow up to the questionnaire during the experiment, an interview was conducted to gather more detailed feedback and better explore feelings, improvement ideas, and pain points. The interview had the goal of generating more feedback as well as discovering any additional requirements the participants had in mind. This was performed as a semi-structured interview where the researchers had the following questions prepared while being prepared to add follow up questions. The main questions for the semi-structured interview can be seen in Table 4.2.

The interview was, together with the rest of the experiment, video recorded to better capture the entire conversation. The interview was then transcribed into notes after the experiment, and the main findings was sent by email back to the parti-

Nr	Question
1	What are your first thoughts about the proposed system?
2	What was the best part of creating drag-and-drop questions in Inspera?
3	What was the worst part of creating drag-and-drop questions in Inspera?
4	What was the best part of creating drag-and-drop questions with the new system?
5	What was the worst part of creating drag-and-drop questions with the new system?
6	Do you feel the system made you more productive?
7	What do you think of the systems GUI?
8	What additional features would you like to see in the new system?
9	What needs to be done for this system to be used regularly by you and other course supervisors?
10	How could the system be made easier to learn and use?
11	What aspects of the system was the most confusing?
12	How could the system make you even more productive and efficient?
13	Do you think there is a difference in the quality of the two solutions you created?
14	Is there anything else you would like to mention or comment regarding the system?

Table 4.2: Interview questions

cipant to check if anything was misunderstood or if they would like to add additional information. As previously discussed for the design and creation strategy, interviews are a better way to discover information regarding requirements rather than ordinary questionnaires, and was thus chosen as a supplement to the already performed observation.

One of the drawbacks of such an interview is that the interview, transcribing, and analysis can be quite time consuming for the researchers. The questions can also be misleading and the answers given can also be misunderstood by the researchers. The questions were created to be as open-ended as possible and to start a conversation rather than creating consistent and direct answers. By accommodating to the abstract nature of a semi-structured interview, one might discover more and new ideas.

Another problem with the interview was that due to it being video recorded, its

intrusive nature might make the situation somewhat stressful for the participants and inhibit honest answers. By conducting these interviews in the participants own offices, the environment could hopefully make room for a more comfortable experience. The video recording might also make some people too uncomfortable to accept an invitation to participate. Since the entire experiment process with observations, questionnaires and interviews took about 45-60 minutes to complete, the duration itself could make the participants tired, annoyed, or inclined not to participate in the experiment.

When conducting an interview, the researchers have to pay attention when the users actions do not conform to their feedback. Reasons for this might be that the user feels pressured to comment, obliged to be nice, or the aesthetic-usability effect (Lidwell et al., 2010). To combat these biases, the interview setting has to be addressed. To begin with, it is important to set the atmosphere as comfortable as possible, and emphasise that the user takes his or her time, and only ask open ended questions if necessary. Second, the researchers should try to distance themselves from the independent variable as much as possible. This entails keeping emotional responses to a minimum, and ensuring the users that none of their responses will hurt any feelings. Lastly, to combat the aesthetic-visibility effect, the researchers might have to ask the users to comment beyond the visual features of the independent variable. However it is important not to ask leading questions, but rather open questions.

4.3 Evaluation

This section discusses the main ways the generated data was evaluated and analysed in the research.

4.3.1 Research Question 1

As discussed in Section 4.1.1, Research Question 1 does not focus on the actual evaluation of the IT artefact since this is the main focus of Research Question 2. The design and creation of the IT artefact still require some iterative evaluation of the development choices made, and this is shown and discussed in Section 5.1.

4.3.2 Research Question 2

This section shows how the different data from Research Question 2, in regards to the evaluation of the IT artefact, was evaluated and analysed in the research. As previously mentioned, the effectiveness of the new proposed system was meas-

ured by comparing the quality of solutions, number of errors made, and questions asked/assistance needed. The efficiency was measured by comparing the task completion time of the pre- and post-tests, and the satisfaction was measured using the SUS questionnaire and a short interview. The quantitative data gathered is here separated from the qualitative data to give a better overview of the different data gathered and how they were evaluated separately.

The null hypothesis stated for this research was:
H0: *The efficiency, effectiveness, and satisfaction scores will be no different between the existing (Inspera) and the new process (IT artefact).*

Quantitative data
All of the data mentioned below are measured once during the pre-test and once during the post-test.

System Usability Scale (SUS):
The SUS questionnaire gathers ordinal data with the use of Likert scale-based questions where numbers are assigned the responses "Strongly disagree"(1) - "Strongly agree"(5). As mentioned in Section 4.2.2, Brook (Brooke et al., 1996) provides a set of rules to calculate a single number between 0-100 representing the overall usability. It is important to view this single number of ordinal data with a critical eye, since there are no way of knowing exactly how much better a score of 80 is than a score of 70.

Errors made:
From the observation, the number of errors made was counted and compared between the pre- and post-test. With this type of data, the number of errors clearly show that 1 error compared to 2 errors is the same as 3 compared to 4, giving a scale with consistent intervals. In the data, 0 errors also have a clear meaning, making this ratio data. A small problem with this ratio data is that it in no ways describe the size of the errors and how severe they are. One large error will still be seen as better than 3 small ones, even though this might not be true. But since categorizing what goes as severe errors or minor errors are highly subjective, the researchers did not find it necessary to reflect the severity of the errors in the quantitative data gathered. But since the consequences of the errors could be quite severe, they were all noted down in detail as qualitative data, so a proper subjective evaluation could be made. The most important thing was to get rid of any critical errors for the final prototype version.

Deciding what goes as an error can be quite difficult to establish, but it is important to clearly define the different kind of errors and inaccuracies that were supposed to be tracked during the testing. The main categories used during the testing was:

- **Errors made during testing (by user)** - These were errors where the participant was in fault and caused an error. These are cases where the user clearly forgot to do something or expected something different to happen. These errors were counted even if the test subject fixed the error at a later point. The reason these errors were counted was to get a better understanding with how easy and how many times misunderstandings occurred. An example of an error when creating a drag and drop task in Inspera would be to forget to pair up a drag area and a drop area.

- **Errors made during testing (by system)** - It is important to distinguish between the errors caused by the user and errors caused by the system in use. This category focuses on bugs, where the software system causes an error, failure, or fault. These errors are also counted when the system produces incorrect or unexpected results and when the system behave in unintended ways. An example could be when the user expects the system to support *copy-paste* functionality, but it does not. Since such functionality is to be expected by most systems, this is categorized as an error by the system, and not by the user.

- **Errors in solution** - These were errors that were present when the test subject stated they were completely finished with the given task. So these are errors that were unknown to the participant and that were not fixed during the test. An example would be to create a 2D Parsons problem without turning on the snap functionality (Turning off "Free placing drag areas") in Inspera as mentioned in Section 2.4.2. Another example would be the lack or misplacing of solution links between drag and drop areas, so that students with a working solution might get points deducted.

- **Inaccuracies** - Inaccuracies was created as its own category to better distinguish between errors in the solution and situations where the functionality was mostly correct, but the visuals were incorrect. A good example of an inaccuracy would be to create a grid of drop-areas where some areas overlapped and some had different sizes. This is not an error, since the grid would still work as intended, but it would be a lot harder for a student to know where to place the drag-area and more visually unpleasant to complete the task. So these were situations where the created solution did not look like intended according to the task sheet the participants were given. A

problem with measuring inaccuracies was that it required a lot of subjective assessment to, for example, decide if an inaccurate drop area grid would count as one single inaccuracy or 30 (one for each drop area). By choosing the latter option, the researchers could potentially cheat their way to better results. So instead of counting every specific inaccuracy incident, the researchers counted the number of inaccuracy categories. This means that the researchers counted a single incident of "Inaccurate drop area sizes" instead of 30 specific incidents of inaccurate drop area sizes. To better strengthen the validity of this measure, all the inaccuracy categories were predefined here:

- Inaccurate drop area positioning and alignment
- Inaccurate drop area sizes
- Inaccurate drag area positioning
- Inaccurate drag area sizes

Questions asked/assistance needed:
As with the errors made, this data is also categorized as ratio data. It also has the same shortcomings as the error data. During the testing, the participants were asked to think aloud and to ask questions if they had any. These questions were noted and counted. It was important to distinguish between questions concerning the actual process that was being tested and the questions that did not. A small question such as *"I was supposed to create all these drag areas, right?"*, would not be counted since the question itself was directed to the understanding of the test, and not the process of creating Parsons problems in either Inspera or with the proposed IT artefact. The questions tracked were questions like *"How do I connect this drag area with the drop area?"*, since these questions were related to the actual systems and processes being tested.

The number of times assistance was needed was also counted, and assistance was only given at times where the participants were completely stuck. Assisting the participants without them asking for help was seen as a last resort and was to avoid if possible during the testing. It was critical to let the participant ask for help before the researchers started intervening, and this was to give the participants a chance to figure it out by themselves, but in some extreme cases, unprovoked assistance would still be needed. An example would be when a participant got completely stuck inside their own file explorer since they did not remember to close or finish the already opened file explorer. The participant thought the entire system crashed and was about to close the system, so in this case the researchers decided to intervene and count this as 'assistance needed'.

Task completion time:
The data gathered from measuring the time it takes to complete a given task have a proportionate interval and a true zero, making this ratio data. The time was only measured during the most relevant and non-overlapping parts of the two processes. As previously mentioned, things such as navigating around in Inspera to get started with the task was not measured.
Since the researchers focused on performing verbal protocol analysis while the participants were performing their tasks, the measured time could be highly influenced by their aloud thinking. It was very likely that the time measurements would be different if a participant were sitting alone and trying to solve the tasks in silence. The verbal protocol analysis leads to extra discussion during the experiment as well as additional feedback regarding improvements and bad practices. When a participant sees or misunderstands something, they spend extra time explaining what is wrong and how it could be improved. This additional time means that the actual task completion time might not reflect 'normal use', but since the verbal protocol analysis influences both the IT artefact tasks and the Inspera tasks somewhat equally, the time measurements was still considered a decent comparison between the two processes.

Qualitative data
The verbal protocol analysis and the interviews held at the end of each observation, as mentioned in Section 4.2.2, was the main source of quantitative data gathered for the evaluation of the IT artefact. Since the entire experiment was taped, video was the main tool for gathering the raw data. The video tape consisted of the users performing their given tasks in the two systems, as well as an interview at the end of the experiment.

From the first part of the video tape, where the users would try to create Parsons problems manually in Inspera and with the IT artefact, any important or unexpected behavior was noted. It is extremely difficult to subjectively decide what is noteworthy and what is not, but in this context, any deviations from the expected behavior or anything that the researchers found interesting was noted. Examples of what counted as interesting was things such as small pauses where the user was waiting for something to happen or got confused as to what a button did. While this kind of evaluation might be inaccurate, the consequences of this inaccuracy was tolerable for this research. It was more important to find the main and the most visible deviations instead of every single one. These unexpected behaviors could then be used as guidance to where additional improvements should be made. If the users showed uncertainty as of what a button would do, the researchers got a good indication that the button needed a more descriptive text, a new icon, better

feedback, or some other informative feature.

The users verbalizations during the test and their activities was recorded using video tapes and a full transcript of the session was made after the testing. Both the verbalizations and the actions they performed were transcribed together to form a complete episode of what was going on.

After the experiment, the entire video would be transcribed and segmented into units. These units would thus consist of verbal protocol analysis episodes, action descriptions, interviews, and other comments. These units were paragraphs in the transcribed texts, and the paragraphs themselves were created with the researchers subjective opinion as to where the answer to a question or a topic ended. To properly perform data analysis, the first step was to identify the key themes in the data. The created units would first have to be duplicated in a new document and placed into a table with two columns. All segments that had no relation to the overall research purpose was discarded in this first step. All remaining segments was given their own row in the table. The remaining units would then be subjected to a theme analysis (Oates, 2005), where each unit or episode were labeled, or coded, with one or more categories and descriptions of the theme presented by in the unit. This made it easier to see which categories and descriptions that were repeated or emphasized by the participants. To properly analyse quantitative data, it is important to look for themes and patterns, and by looking at the connotations, i.e. the values and ideas expressed, and finding what categories and descriptions are repeated, one get a better understanding of what might be considered as important. To develop these categories/coding schemes, one can choose between a deductive approach and an inductive approach. With the deductive approach, one develops the categories beforehand and create these categories based on literature or theories. For verbal protocol analysis, deductive a priori coding schemes are recommended to facilitate efficient analysis of the data (Todd and Benbasat, 1987). The inductive approach, on the other hand, lets the researchers find the categories from the data itself as one goes through the material. In this research, the latter was chosen to avoid committing too much to already created theories. An inductive approach might be somewhat better for discovery and might make it easier to avoid overlooking new and other themes in the data. A problem with the inductive approach is that the categories may be highly influenced by the researchers previous thoughts and experiences, but completely avoiding this kind of subjective prejudices can be difficult in quantitative analysis, and to avoid researchers bias, the coding should preferably not be performed by the researchers themselves at all. Since the research already consists of other predefined scoring schemes in the quantitative data gathering, the inductive a posteriori coding scheme was chosen for the interviews and verbal protocol analysis. While this might be more time-

consuming and be influenced by researcher bias, acknowledging the shortcomings might help the researchers approach the categories a bit more open minded. Since the quantitative data focused more on discovery, the categories created by the researchers naturally focused on the more negative aspects of the feedback in order to find where improvements could be made.

As mentioned, the goal of the interview and the verbal protocol analysis evaluation was to discover additional feature requests or shortcomings and to better understand how the IT artefact was perceived by the users. Together with the categories created, the feature requests mentioned by the participants during the testing and interviews was grouped together and sorted by how frequently they were mentioned during the testing. This list was the main source for further development of the IT artefact.

4.4 System Development Method

This section discusses the system development method used in the development process of this IT artefact.

4.4.1 Agile Software Development

Because the domain of creating exams was fairly unknown to the researchers, and since the initial requirements were somewhat flexible, a plan-driven development method like the waterfall model would not be appropriate. Rather, it was natural to implement an incremental, agile systems development method to get continuous feedback and build domain knowledge.
Agile development methods embraces change, and thrives in small development teams developing small or medium-sized products (Sommerville, 2011). As the initial requirements were expected to change throughout the project, and the development team consisted of two programmers, the agile methodology was seen as the best fit. In addition, the thesis supervisor could arguably be seen as the project customer, such that customer involvement in most of the development process could be guaranteed, which is highly appreciated in agile development methods.

There are many iterative models available, of which many would have been a good choice, but due to the small size of the development team, it was not seen as necessary to adhere one specific model. With two developers developing side by side throughout the entire project, communication was considered to be highly efficient, which opened for the possibility of using elements from different methodologies that best suited the development process.
It was decided to utilise agile elements from the Extreme Programming, Prototyp-

ing, and Kanban methods. The development cycle was to have an initial stage of requirement gathering, iterations to incrementally improve a prototype, and use the final prototype as a release version. The release cycle from Extreme Programming, combined with prototyping, was seen a good fit for the incremental development. In addition, pair programming was utilised from Extreme Programming, and the kanban board from the Kanban development method.

Development Structure

An overview of the development process was as follows:

1. Identify basic requirements
2. Initial design of architecture
3. Continuously develop a prototype through iterations until satisfactory end product
    - User testing in Iteration 4
4. Final release

Before the iterations commenced, an initial set of requirements were gathered, which would allow for an initial design of the application architecture. As this was a preface to the first iteration, it was labeled as Iteration 0 to emphasize that this was done prior to any development on the IT artefact.

When an initial architectural design had been provided, the first iteration could commence. Incrementally, each iteration cumulatively build on the prototype until a satisfactory version of the IT artefact was produced.
In Iteration 4 it was scheduled to do user testing of the IT artefact. For this reason, it was important that a testable prototype with the most valuable features was ready.

After the final iteration, the IT artefact was seen as ready to be released.

Iteration Structure

The iteration structure was based on the release plan provided by the Extreme Programming method (see Subsection 4.4.2). At the beginning of every iteration, requirements were picked and broken down into smaller tasks. The requirement were gathered from the initial list of requirements, feedback from user testing, and decisions made by the developers themselves. The reason for breaking down the

requirements into smaller tasks was to have more specific targets to work towards. Additionally, smaller tasks were easier to describe with a 'definition of done'.

After each iteration, the IT artefact was evaluated by doing an assessment of its worth and deviations from expectations. The research questions of the thesis were concerned with how to create and design a system for efficient generation of Parsons problems, and the effects of using it - compared to the manual method. For that reason, the evaluation criteria was mainly usability, functionality, and aesthetics.

4.4.2 Extreme Programming

Extreme Programming (XP), created by Kent Beck (Beck and Gamma, 2000), takes iterative development to 'extreme' levels, and provides useful techniques for development. In particular, two elements from XP were included in this development process: the release plan and pair programming.

The release plan is broken into six phases that are looped until a satisfactory product is produced (Sommerville, 2011): select user stories, break down user stories into tasks, plan release, development, release, and evaluate. For the development process of this IT artefact, the release plan was tweaked a bit. There were no user stories, instead tasks were broken down from requirements. The reason for this was that the thesis supervisor (the customer) had great technical knowledge, and therefore technical requirements sufficed. At the end of each iteration, the prototype was presented to the supervisor for feedback, and thereafter evaluated.

The steps of the implemented release plan were as follow:

1. Select requirements for the iteration
2. Break the requirements down into tasks
3. Develop the prototype
4. Supervisor feedback
5. Evaluate the prototype

Pair Programming was utilised throughout the development process, as it entails benefits like collective ownership of code, fewer bugs, and sharing of knowledge (Williams et al., 2000). But there are also pitfalls to be wary about - for example troubles with team pairing and difference in skill level. Some of the challenges can be traced back to communication. But in this case, communication was not

seen as a challenge, but rather a strength. The developers had worked together on several projects in the past, and were situated on a desk next to each other, which makes for a perfect opportunity for pair programming.

4.4.3 Software Prototyping

In combination with the release cycle from XP, it was natural to use the prototype method for developing the IT artefact. The reason for this was that prototypes are useful when the end product is not clearly stated, especially when it comes to the user interface. Prototypes allows for exploration of feature possibilities, which was highly beneficial as the requirements were somewhat flexible.

When prototyping, there are two dimensions in regards of development (Nielsen, 1994): Horizontal prototyping and vertical prototyping. The horizontal dimension focuses development on all features and branches at the same time, while the vertical dimension focuses on fully implementing one feature at a time.
Developing the IT artefact, it was decided to opt for the horizontal prototyping dimension. As there was a lack of requirements for the user interface, it was not given which features were to be implemented. Therefore, it would be most beneficial to implement many features, and reducing the level of functionality of each feature, to see how they play together as a whole. If one were to go for the vertical dimension, where entire features are implemented with functionality, there was a risk that some of the implemented features would not make it to the final version of the prototype, and time would be wasted. Opting for the horizontal dimension was considered safer, and it allowed for the whole system to be tested from the very beginning, resulting in earlier feedback on the system as a whole.

There are several models one can adhere to when prototyping: Throwaway prototyping, evolutionary prototyping, or operational prototyping (Davis, 1992). Because of the length of the development period (two semesters), it allowed for the evolutionary prototyping model. This way of prototyping involves continuously developing one prototype throughout the development process, instead of discarding prototypes after each iteration.
As the prototype evolved through the iterations, the final prototype was a result of continuous refinement and improvement of source code, and was therefore considered to be a complete, final version of the IT artefact.

A disadvantage with evolutionary prototyping is attachment to the prototype. When spending a great deal of time on a prototype or feature, it is easy to get overly attached to that particular subject, and as a result include features that are not necessary. This was something that the developers had to keep in mind during de-

velopment.

4.4.4 Kanban

Supplementing the software development process, it was decided to utilise a kanban board from the lean development method Kanban (Sugimori et al., 1977). Having columns for each stage of implementation, visualisation of the overall process of development was made easy. Kanban boards play well with the horizontal approach to prototyping, as it is easy to get lost when you have many irons in the fire at the same time. Trello[1] was chosen as the tool for implementing the kanban board.

4.4.5 Testing

The process of creating exams have to be reliable. If the tasks generated by the IT artefact have incorrect solutions, students who answer correctly will not be awarded points, and student answering incorrectly might be. If tasks are not generated as intended by the end-users, tasks might end up making no sense, be insolvable, or in conflict with Inspera. If the IT artefact encounters bugs during use, tasks might not be generated correctly. To prevent these situations, and reinforce reliability, the IT artefact had to be properly tested.
Over the course of development, three stages of testing were implemented: development testing, system testing, and user testing. The goals of the testing was to demonstrate that the IT artefact met its requirements, and to discover situations in which the behaviour of the IT artefact was incorrect, undesirable, or did not conform to its specifications (Sommerville, 2011).

The development testing was carried out during the development of the IT artefact and consisted of unit testing. As the essence of the IT artefact was the back-end of the code (the QIT Converter), it was the focus of testing, and required the widest test coverage.

The system testing was carried out by doing a set of use cases, where each use case had an expected outcome. This stage was mainly done after implementations of each requirement, before the user testing stage, and before the final release of the IT artefact. The use cases were annotated with steps for execution, and the expected result for verification. The goal of the system testing was to demonstrate that components worked together as expected, which entailed both testing components within the IT artefact and testing the interaction between the IT artefact

[1] https://trello.com/

and Inspera.

The user testing stage was responsible for validating the IT artefact. That is, making sure that the system provided expected behaviour, and conformed to user expectation. In addition, user testing allowed for discovery of situations where the behaviour of the IT artefact was incorrect and/or undesirable, and provided a basis for feedback on future improvements. User testing is further described in Section 4.2.2.

Even though the IT artefact was thoroughly tested, there are still no guarantees that all bugs and inconsistencies were found. For this reason, we emphasise that end-users verifies the generated tasks in Inspera before any use.
As Edsger Dijkstra stated in his article "The Humble Programmer" (Dijkstra, 1972):

> *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."*

Chapter 5

Results

In this chapter, the results obtained will be revealed. First, the results from the design and creation strategy, with iterations and the topics permutations, technology choices, and design are displayed. Second, the quantitative and qualitative results from the experiment strategy are presented.

The resulting IT artefact can be found at
https://github.com/joachimjorgensen/masteroppgave-js.

5.1 Design and Creation

Research Question 1 was concerned with the design and creation of an IT artefact for efficient generation of Parsons problems for digital programming exams. In this section, the results regarding Research Question 1 is outlined. The section is separated into subsections for every iteration, to display the evolution of the IT artefact throughout the project, and subsections for Technology Choices, Permutations, and Design. The development of the IT artefact spanned five iterations, and corresponding subsections (except Iteration 0) are structured as follows:

Goals: Provides a list of the main requirements that was worked on during that period, the specific tasks created from these requirements, and the overall goals of the iteration. The tasks are specified separately from the requirements since they were the more specific assignments from the developers kanban board.

Implementation: Discusses how the requirements, tasks, and features were implemented and why.

Testing: Discusses the testing stages of the iteration, and lists the use cases performed for system testing.

Evaluation and feedback: Provides an overview of the feedback gathered from each iteration, as well as an evaluation of the iteration and the implemented features.

5.1.1 Iteration 0

Iteration 0 was defined as the pre-development stage, and mainly consisted of gathering information and planning. The main goals of this iteration was to identify requirements for the IT artefact, design models of the architecture, and get a better understanding of what to create.

How the end-user requirements were gathered is described in Section 4.2.1, and a complete list of the gathered requirements, as well as the non-functional requirements, can be found in Appendix D.

When developing the initial architecture of the IT artefact, it was still somewhat unclear what technologies to utilize. Section 5.1.8 describes the thought process behind the choices made in regards to which technologies to utilize. At this time in the development process (Iteration 0), it was decided that creating a desktop application using Electron would be the best solution. The front-end of the application was to be written in pure JavaScript, while the back-end consist of a Flask API connected to a QTI generator (Hereinafter referred to as QTI Converter) written with Python.

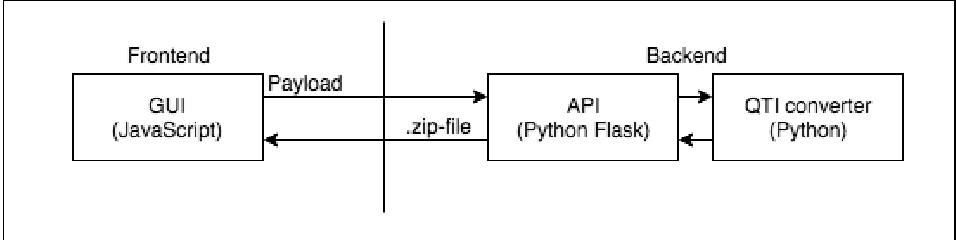

Figure 5.1: System Architecture v1.0

Figure 5.1 shows the main relations between the initial components, and Figure 5.2 shows the sequence diagram for the IT artefact. In its simplest form, the user

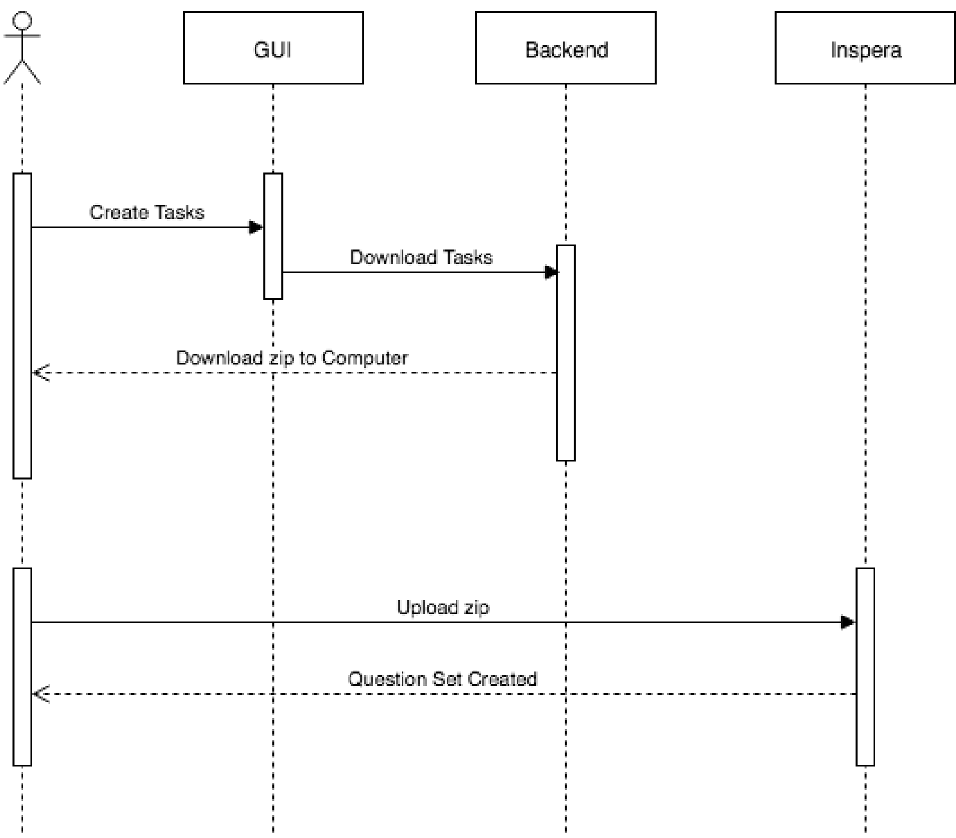

Figure 5.2: Sequence Diagram

should be able to upload their code snippet to the IT artefact, and thereafter export it as a zip-file that can be uploaded to Inspera as a drag and drop task. This zip-file should require no extra modifications and be ready for use. The sequence diagram and the system architecture are both quite simplistic, and this simplicity was valuable when creating a prototype that required testing as soon as possible. The entire project was now clearly structured into the creation of a GUI that let the users upload code and fill out the required settings, and a QTI Converter that could create tasks ready to be uploaded to Inspera. In Appendix A Section A.1, one can also see an activity diagram of the assumed activities at this point. The clear separation between front-end and back-end also made it possible to develop these parts in parallel. Single requirements or tasks could then be implemented in the GUI and the QTI Converter at the same time.

Immediately following the gathering of requirements, the next step was to create a simple mock-up of the GUI of the IT artefact. This gave a better understanding

of what the different requirements would require of the application, and creating a mock-up lets you quickly test out multiple design options. Figure 5.3 shows the first application concept with its minimal design, only taking core functionality into consideration. The design was created in Photoshop[1] for easy manipulation of ideas and suggestions that came to mind. Design theory is described in Section 2.7, and will be further discussed in Section 5.1.9.

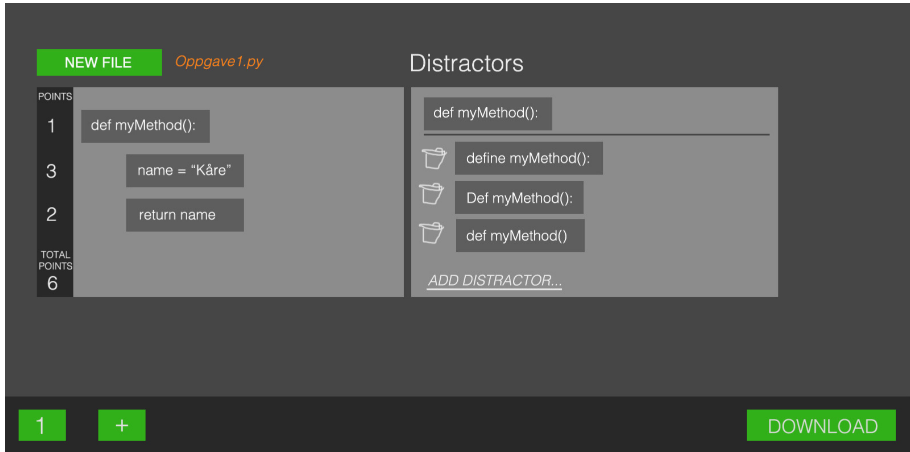

Figure 5.3: Mock-up design from Iteration 0

The idea behind the first mock-up was to segregate the different tasks by implementing a tab functionality, rendering navigation self-evident. Assigning each tab a number at the bottom of the application, in ascending order, made for easy tracking of all tasks.

The task page was initially built up of three components: a file uploader, source code editor, and distractor input. The file uploader was placed top left, because it is advantageous for the user to notice this functionality before the source code editor, to avoid possible duplicate input. Below the file uploader, the source code editor was placed, as this was naturally the second functionality of focus. The source code editor was utilized with a point marker for each line of code, to support a point scheme for each task. Lastly, the distractor input was placed to the right of the source code editor, to suggest a connection between them.

[1] https://www.adobe.com/no/products/photoshop.html

5.1.2 Iteration 1

Goals

Functional Requirements for Iteration 1

ID	Title	Description
6	Generate Inspera supported QTI	The Parsons problem to be generated by the IT artefact must be generated in a QTI format that is supported by Inspera

Tasks for Iteration 1

Title	Definition of done
Initialise Electron Application	Create an empty Electron Application ready for use.
Initialise Flask API	Create an empty Flask API ready for use.
Create QTI Converter	Create the first version of a Python program that uses the QTI skeleton code and adds the required fields for creating a drag and drop task.

The focus of this iteration was to develop a working version of the QTI Converter seen in the back-end section of Figure 5.1, while getting started with the initial setup of the Electron application. More precisely, the main goal was to be able to generate a working drag and drop task with the QTI-format, compatible with Inspera. This functionality was the essence of this IT artefact, and therefore, setting up this core functionality as quickly as possible was paramount.

Implementation

At this point it is important to mention that Inspera supports both the importing and exporting of questions in QTI format. This meant that instead of building the entire QTI file from scratch, and risk forgetting some required XML fields, one could export an empty drag and drop task from Inspera and use this as a skeleton for all future task to be generated. The QTI Converter was at this point written in Python, which has multiple libraries for parsing and working with XML, making it easy to add additional fields to the mentioned skeleton. Figure 5.5 shows the function that parsed the XML skeleton string (but here written in JavaScript). The long XML file was formatted to a single-lined string to save space. While using this skeleton XML made the initial development easier, it could lead to some unnecessary complications in the future. If Inspera at a later point updated to a another version of QTI or changed the requirement of some specific fields, it could

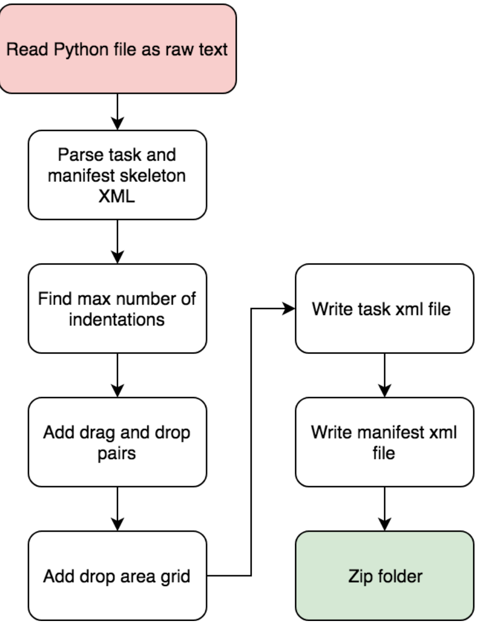

Figure 5.4: Flowchart of QTI Converter in Iteration 1

be harder to debug and fix the skeleton code than if the entire XML was built from scratch. Since the main focus during this iteration was to make the task generation work, these ’shortcuts’ were acceptable.

```
/**
 * Parses a skeleton drag and drop task text into a XML document.
 * The task text is an empty drag and drop task generated by Inspera in QTI 2.1.
 *
 * @return {XMLDocument object} An empty drag and drop XML document
 */
function read_xml() {
    let text = '<?xml version="1.0" encoding="UTF-8"?><assessmentItem xmlns="http://www.imsglobal.org/xsd/imsqti_v2p1" xsi:sch
    let parser = new DOMParser();
    let xmlDoc = parser.parseFromString(text, "text/xml");

    return xmlDoc;
}
```

Figure 5.5: Parsing the skeleton XML in JavaScript

To keep it simple, the program was executed through a command line interface where it read a single Python file as raw text and created the drag and drop task. A flowchart of the initial QTI Converter code can be seen in Figure 5.4. The idea was to create working versions of each module in the architecture before trying to connect them. So while the first version of the QTI Converter was created, the Electron application and Flask API was initialised.

Testing

As this was the first iteration of development, not too much functionality could be tested. The Electron application and the Flask API had no extra functionality other than what was included in the setup guides, while the development of the QTI Converter was still in the stages of figuring out the correct way of creating the exported files. For this reason, use cases were mainly used for testing the QTI Converter, instead of using unit tests, since the exported files had to be manually tested against Inspera.

Use cases can be found in Appendix C.1.

Evaluation and feedback

At the end of Iteration 1, the first version of the QTI Converter successfully generated a simple drag and drop task given text from a single Python file. At this point it generated a 2D Parsons problem consisting of a grid of drop areas and drag areas placed at fixed coordinates. These results were presented to the thesis supervisor at the end of the iteration. Since there existed a detailed course of action, the feedback given proposed no changed to scheduled plan, current implementation, or requirements.

This iteration proved that the proposed system could work as intended, and could be further developed. There were no unexpected happenings or difficulties that occurred during this iteration. The iteration was concluded as a success.

5.1.3 Iteration 2

Goals

Functional Requirements for Iteration 2
(Note that these are the new requirement added for this iteration. Previous requirements were still relevant)

ID	Title	Description
1	Desktop application	The IT artefact should be a desktop application providing the users with a GUI where one can perform all necessary actions to properly generate Parson problems

2	Parsons problems and 2D Parsons problems	The IT artefact has to support the generation of both 1D Parsons problems as well as 2D Parsons problems
3	Upload file	Users should be able to upload any code file to the IT artefact, such that this code can be used as the basis for the Parsons problem to be generated
4	Copy and paste source code	Users should be able to copy and paste source code into the IT artefact and use this source code as the basis for the Parsons problem to be generated
5	Write and edit source code	Users should be able to write and edit source code in the IT artefact and use this source code as the basis for the Parsons problem to be generated
7	Parsons problems design	The Parsons problems to be generated by the IT artefact must follow the predetermined design of how Parsons problems should be created in Inspera using drag and drop tasks
11	Export all and **Export single**	When creating multiple tasks, the user should be able to export both a single task and all tasks (Focus on exporting a single task this iteration)
14	Export destination	The user should be able to select the exact folder the exported task should be placed in when exporting from the IT artefact

Tasks for Iteration 2

Title	Definition of done
Add code editor	Add a code editor to the Electron application, making it possible to write, copy and paste code
Upload file	Make it possible to upload a file
Display content of uploaded file	The uploaded content should be transferred to the code editor in the Electron application
Export task	A user should be able to export/download the generated task (.zip-file) through the GUI
Export destination	Make it possible to choose the name and folder destination for the generated task (.zip-file)
Continuous saving	Instead of a designated Save button, make the Electron application save its current state on change
Random drag area placement	Make the generated drag areas appear in a random order in Inspera
Normal Parsons problem and 2D Parsons problem	Make the QTI Converter support the generation of both 2D Parsons problem and normal Parsons problem
Connect front-end to back-end	Have the front-end send its state to the Flask API, which in turn makes the QTI Converter generate a zip-file
Packaging/build Electron application	Create a release build of the Electron application with all the components in the architecture that is executable for both Windows and Mac OS

The goal of Iteration 2 was to have a minimalistic, functioning user interface communicating with the back-end. For this reason, the apparent design aspect was not highly prioritised, but rather the inherent usability. Additionally, the QTI Converter was expanded to support the generation of 1D Parsons problems as well as 2D Parsons problems.

At the beginning of this iteration, some functional requirements were discarded and some were added. Appendix D Table D.3 shows some of the removed requirements, where many were removed during this iteration. Table D.3 ID 1 (Suggest distractors) was discarded due to the IT artefact being an offline desktop application. The idea was to use machine learning or some predefined set of rules to generate distractor suggestions for each line of code given by a user, but without a connection to the Internet, the provided value from this feature would be low compared to the effort of implementing it. Table D.3 ID 2 (Comment syntax) were mentioned quite a lot during the initial requirement gathering, but during this itera-

tion it was decided to only create a GUI instead of having to support multiple ways to perform the same actions. Using a GUI instead of a comment syntax could also support scalability, ease of use, and usability since it can provide better feedback, tutorials, and help methods. Table D.3 ID 3 (Score allocation) was also discarded because it was discovered that these kinds of score allocations was impossible to create with Inspera.

During Iteration 2, some new requirements were added to the list as well. The new requirements are shown in Appendix D Table D.1. Table D.1 ID 13 (Preview) was added as a feedback mechanism that could help the user visualise and understand what kind of task that would be generated before exporting it. Without a preview function, the user would not see if an error had been made before exporting the task from the IT artefact, uploading it to Inspera, and previewing it there. Table D.1 ID 14 (Export destination) was a small and obvious feature previously overlooked and simply forgotten to specify. Table D.1 ID 15 (Help functions) was added to the list of requirements to help new users of the IT artefact learn and understand it.

Implementation

Adding a code editor to the Electron application was easily done by utilising CodeMirror[2], which is a JavaScript component providing a code editor for the browser. With its extensive API, uploading files and displaying the code in the editor could easily be handled as well. During this iteration, an option to toggle the 2D Parsons setting was implemented, which was done through a simple toggle button with a label. The tab system mentioned in the mock-up was not yet implemented, only making it possible to create one task at a time. The Electron application GUI for the second iteration can be seen in Figure 5.6.

The front-end had its own internal state, structured as a JSON object, that could easily be transported as a payload to the back-end when exporting the task. The internal state was updated by multiple onChange-methods and saved after every action that provided new information to be stored. The main idea behind this continuous preservation of the internal state was to better accommodate for the tab system mentioned in the mock-up that was to be implemented at a later point and the live updating of the preview.

Making the QTI Converter create both 2D Parsons problems and 1D Parsons problems required some restructuring of the initial code, but was in general easily implemented. The main difference between creating these two task types were that the 1D Parsons problems only consisted of drag and drop area pairs. This means

[2]https://codemirror.net/

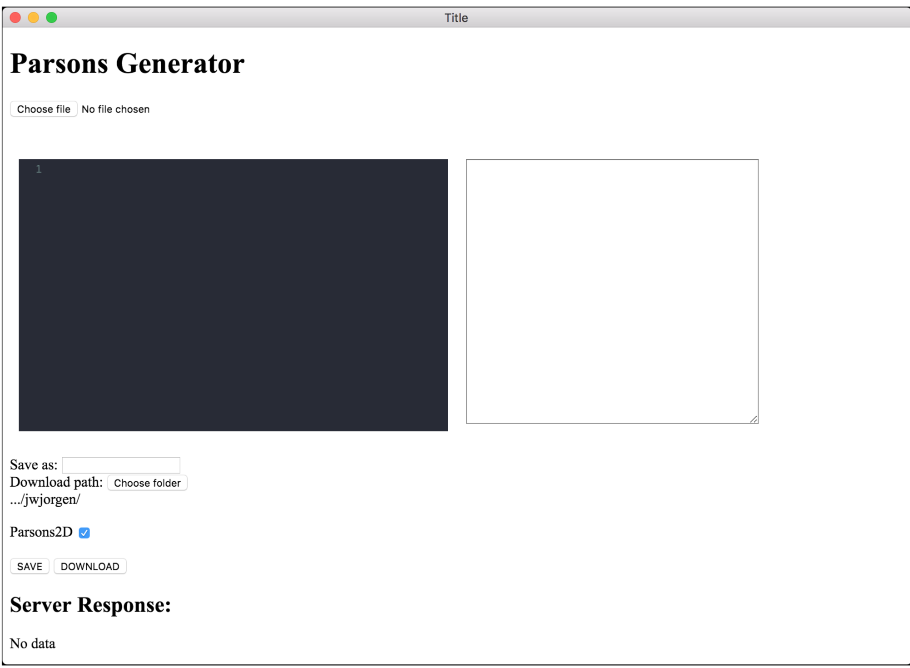

Figure 5.6: Application after Iteration 2

that every drop area has a drag area connected to it, while the 2D Parsons problems has to generate a drop area grid where many of the areas do not have a connection to a drag area. Supporting the random placement of drag areas, on the other hand, was not as easy to implement. It was not possible to have fixed positions that could be randomly distributed to the drag areas since the width of each area varied greatly. Since the QTI Converter created the drag and drop pair sequentially for each line of code, the specific coordinates for the drag areas had to be determined before creating the pair. The solution was to have a separate function generate all starting positions for the drag areas and place these coordinates in a dictionary that could be used when creating the drag and drop pair. To place the drag areas as close to each other as possible, one also had to calculate the exact length of each drag area using specific font metrics and margins from Inspera. Figure 5.7 shows the updated flowchart of the QTI Converter code. The QTI Converter was also updated to support additional metadata such as canvas height, so longer Parsons problems could be created, and title, so a task name could be set in the IT artefact.

During Iteration 2, the Flask API successfully connected the front-end to the back-end. The internal state of the front-end got transferred to the Flask API, which in

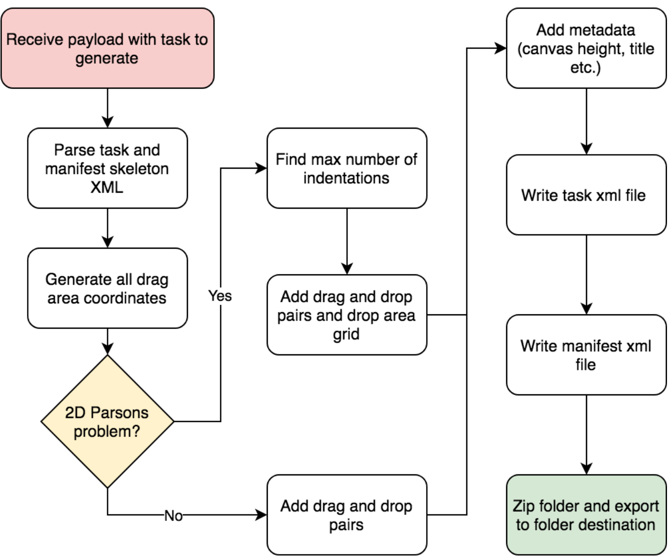

Figure 5.7: Flowchart of QTI Converter in Iteration 2

turn called the proper function from the QTI Converter. But once this functionality was implemented, packaging the IT artefact into a desktop application proved to be more problematic than anticipated. The initial idea of separating front-end and back-end code, and have them communicate, required the IT artefact to open a port and listen for incoming messages (acting like a server). Doing this in developer mode was not an issue, but once the program was packaged to a release build, using Electron, access to the underlying system proved to be troublesome. In addition, running Python scripts from the Electron environment was not an easy task. After some time trying to fix these issues, the developers decided to change and rewrite the entire back-end functionality to JavaScript (which is what Electron expects) instead of Python. This removed the server behaviour entirely, and had the front-end communicate with the back-end directly through function calls. The updated system architecture in Figure 5.8 shows that the Flask API was removed and pure JavaScript was now used in the entire desktop application.

Testing

As the second iteration progressed, more code was added to the project, ready for testing. Back-end code was tested with automated unit tests (development testing), while front-end features and requirements were tested with use cases (system testing). Use cases were also used to make sure communication between the front-end and back-end was correctly executed, and that the correct tasks for Inspera were

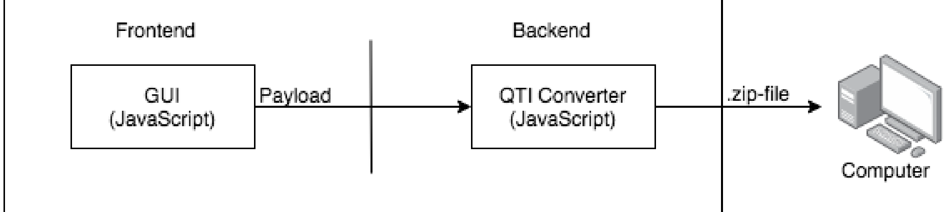

Figure 5.8: System Architecture v2.0

generated.

There were troubles passing the use case regarding the packaging of the application, as the application had problems running the Python Flask server from an executable file. To pass the use case, the application was rewritten in JavaScript, and the Flask server was discarded, as stated in the previous subsection.

Use cases can be found in Appendix C.2.

Evaluation and feedback

Most features were successfully implemented during Iteration 2. And while the packaging issues resulted in having to completely rewrite the back-end, the IT artefact worked perfectly after the change was made. The current prototype was presented to the thesis supervisor for feedback. The feedback was mostly positive and few changes were made to the existing plan, list of requirements, and implemented features. It was during this evaluation meeting the requirements from Table D.3 ID 1 (Suggest distractors) and ID 2 (Comment syntax) were removed, while the idea of having multiple help functions (Table D.1 ID 15) were added.

The second iteration was overall a success, as all goals of the iteration were fulfilled to a satisfactory level.

5.1.4 Iteration 3

Goals

Functional Requirements for Iteration 3
(Note that these are the new requirement added for this iteration. Previous requirements were still relevant)

ID	Title	Description
10	Generate multiple tasks	The IT artefact should be able to generate multiple tasks (of different algorithms) at the same time. It should be possible to create, edit, and delete these tasks
11	Export all and Export single	When creating multiple tasks, the user should be able to export both a single task and all tasks

Tasks for Iteration 3

Title	Definition of done
Improve design	Improve the overall design and aesthetics of the GUI (Need not be fully done, as the design will be updated over all iterations)
Task tabs	Create a tab system for changing between tasks
Edit, create and delete multiple tasks	During the life cycle of the application, one should be able to add, delete, and edit tasks
Generate multiple tasks in one Question set	The QTI Converter should be able to receive multiple tasks in one payload, generate them all, and place them in the same Question set

The main goal of Iteration 3 was to be able to generate multiple tasks at once. This required the addition of the tabular functionality showed in the initial mock-up from Iteration 0 and some restructuring of the QTI Converter.

Implementation

The IT artefact had a major uplift design wise from Iteration two to three. A tab system was implemented to keep track of the different tasks that a user created, but instead of following the initial mock-up design, the tabs were placed vertically on the left hand side of the GUI. The reason for this, was that each task had the option to be assigned a title, making it even easier differentiating them. In addition, in the event of an enormous amount of tasks, it is easier to scroll vertically, than horizontally. See Figure 5.9. Creating a new task would simply create a new object for the internal state of the front-end, and the IT artefact would either send the single object or all task objects when the user pressed 'Download Task' or 'Download All'.

The GUI for each task were divided into two sections (The top and bottom section seen in Figure 5.9), the top section for main task activities such as title, file

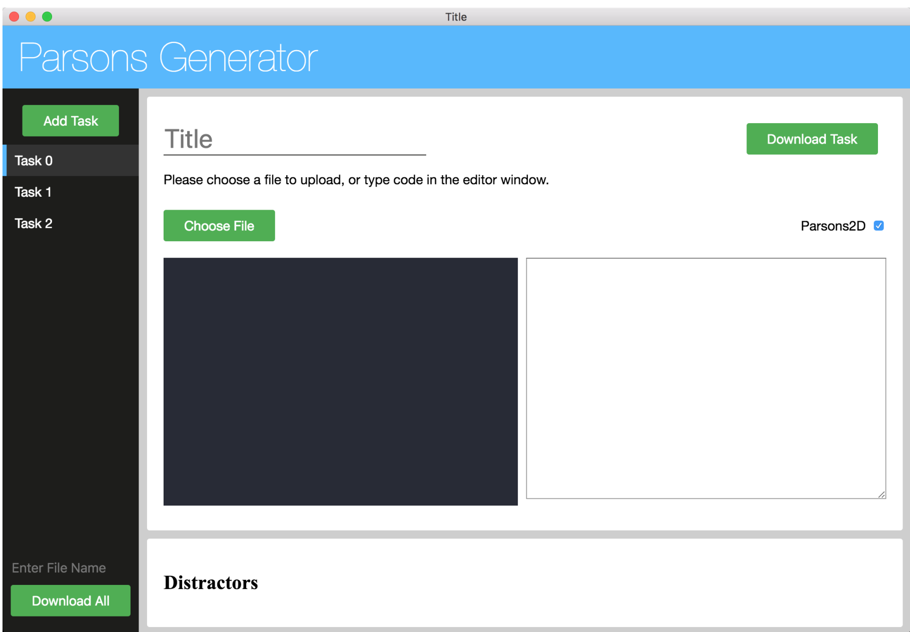

Figure 5.9: Application after Iteration 3

upload, source code editor, parsons 2D toggle, and a container for an eventual preview functionality. The bottom section was reserved for distractors, but the internal structure was not yet decided. And since both the preview and the distractor functionality was not to be implemented yet, the GUI were simply given empty sections where these features would be placed in the future.

The color scheme changed from the mock-up as well, so that the tab system to the left was decorated with a dark background color, and the task background white. This to make a definitive border between the two functionalities. The overall color theme was also changed to a lighter combination instead of the dark and gloomy color theme from the mock-up. More information regarding the design can be found in Section 5.1.9.

The QTI Converter had to handle receiving both single task objects and objects consisting of multiple task objects. To handle the generation of multiple tasks, the QTI Converter had to generate an XML file for each task and make the appropriate changes to the manifest file for them to be generated as a single question set. The updated flowchart for the QTI Converter in Iteration 3 can be seen in Figure 5.10.

Discovering and learning what fields and functions to add or change in the XML

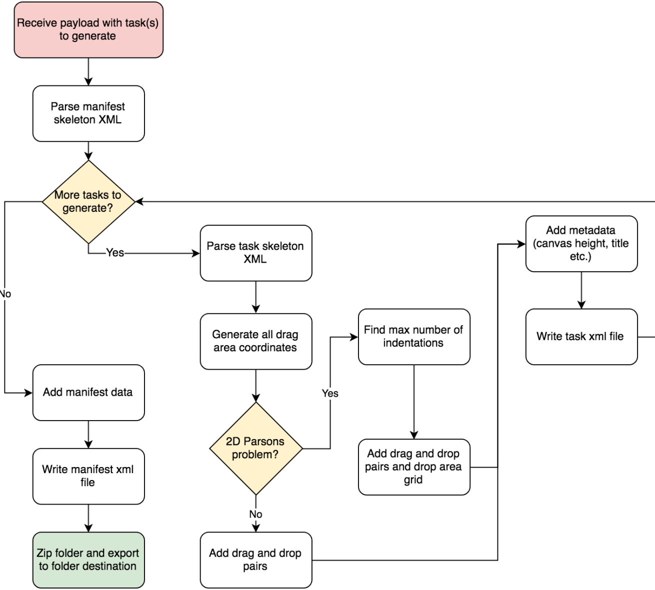

Figure 5.10: Flowchart of QTI Converter in Iteration 3

when adding new functionality, was difficult to do efficiently. The most efficient way found, was to first manually create the wanted behavior using Inspera, export this task or question set and compare the given XML to the currently generated XML. The main reason this method was used, was that the QTI format itself supported several features that Inspera did not. Having to look through and test all QTI features, fields, and functions in the hope of them being supported by Inspera would be much more time consuming than simply creating the wanted behavior in Inspera, and manually see how these changes were reflected in the XML. This method was still quite time-consuming since the generated XML files could be extremely large, but it made certain that all fields added or changed in the generated XML were fully supported by Inspera.

During the iteration, nothing more than minor bugs and challenges were encountered. These were bugs such as the front-end updating the wrong state object and the QTI Converter not properly updating the manifest when multiple tasks were generated. All these bugs were fixed during the iteration and everything worked as intended in the end.

Testing

Testing wise, the iteration had no difficulties. Unit tests were added while developing, and use cases were tested whenever features were implemented.

Use cases can be found in Appendix C.3.

Evaluation and feedback

Both the developers and the thesis supervisor were satisfied with the new proposed design of the IT artefact. The creation of multiple tasks also worked as expected. A user could now create and edit multiple tasks at the same time, and export both a single and all tasks to a zip-file. There were still no need to manually save the progress while using the IT artefact, since all changes made in the GUI were instantly saved to the front-ends internal state.

Iteration 3 was concluded a success.

5.1.5 Iteration 4

Goals

Functional Requirements for Iteration 4
(Note that these are the new requirement added for this iteration. Previous requirements were still relevant)

ID	Title	Description
8	Support distractors	The user should be able to add distractors to the Parsons problems
15	Help functions	The IT artefact should provide the user with informative and detailed help functions to clearly communicate what features do and how things work
17	Confirmation, warning and error messages	The IT artefact should provide the user with proper confirmation, warning and error messages. For example, when deleting a task, a confirmation box should be displayed to avoid extra work if the button was pressed by accident

Tasks for Iteration 4

Title	Definition of done
Add distractors	A user should be able to add and remove distractors to every task using the GUI
QTI Converter distractors	The QTI Converter should randomly place the distractors together with the other drag areas without connecting them to any drop area
Delete tasks	A user should be able to delete a task in the GUI (Provide a warning message before deleting)
Confirmation, warning and error messages	The IT artefact should provide the user with a confirmation message once the exported tasks has been zipped and placed in the destination folder, or an error message if something goes wrong
Help functions	Create a main Help window showing the main steps through the IT artefact. Add additional hoverable help functions for headlines that might be difficult to understand with no previous experience
Name zip-file	After pressing the download button, in the file dialog, one should be able to name the zip-file to be generated
Ready for testing	The IT artefact should have all the necessary functionalities implemented and ready for user testing
Package/build release versions	The IT artefact must be packaged/built into a working release build for both Windows and Mac OS

The main goal for Iteration 4 was to have a prototype ready for more comprehensive user testing. It was at the end of this iteration the main experiment discussed in this thesis (Section 4.1.2) was scheduled. This required extra focus on improving the IT artefacts ease of use and feedback from actions, such that new users could easily understand and learn how to use it. More information regarding the scheduled experiment can be found in Section 4.1.2, 4.2.2 and 4.3.2.

Implementation

Getting the IT artefact ready for proper user testing also required the removal of features not yet implemented or that were not going to get implemented during this iteration, such as the empty section reserved for the task preview. Preparing for user testing was quite time-consuming, and required both planning of the actual test process and fine-tuning of the IT artefact.

Deciding how to implement distractors for the front-end required some extra thought

since it could be done in multiple ways. The three main options were either to (1) have a separate code editor where each line of code would become a distractor in the task, (2) have a designated input-field where one can add one distractor at a time, and (3) create a commenting syntax for distractors in the existing code editor. The commenting syntax was discarded as it was not considered especially user friendly, as the user would have to learn and adhere to the specific syntax for this IT artefact. The two remaining options were considered to be equally viable options, and in the end, the designated input-field for the distractors was the chosen solution. In Figure 5.12 one can see the distractor input-field at the right side of the source code editor. When added, the distractors would then appear in a list below the input-field.

In Figure 5.12, one can also see that '?'-signs were added behind some headers and text-fields. These were hoverable icons that displayed a help popup providing some more information about the specific features. For example, when hovering the question mark icon behind 'With indenting', Figure 5.11 would pop up to give the user a better understanding of the feature. It was later discovered that some of these popups were not as helpful as expected, and this will be discussed later.

As mentioned, it was crucial to make sure that all implemented features of the IT artefact were ready for user testing during the iteration. One of the obstacles encountered was that when packaging and building a release version of the IT artefact, multiple errors related to the file pathing appeared. It was discovered that file pathing was completely different for Windows and MacOS, and completely different for the development build and the release build of the application. The build version of an Electron application, with all its relations and dependencies, were by default structured completely different than the development version. Initially, the feature regarding choosing a file to upload or export had to be implemented differently for these versions, but after switching from the standard HTML file input style to Electrons build in file input, the implementation of both the build version and release version were same. The difference in pathing styles in the operation systems were still an issue, which was solved by making all paths change based on a conditional statement checking for operating system type.

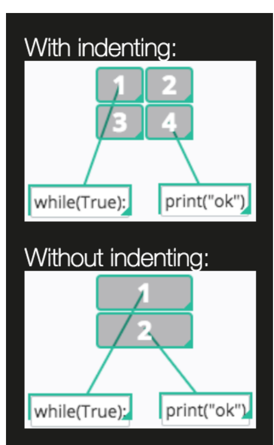

Figure 5.11: Help popup for 'With indenting'

During Iteration 4, no large changes were made to the flowchart of the QTI Con-

verter from Figure 5.10. The main difference was that when adding the drag and drop pairs, all distractors were added as well.

The user testing was conducted at the end of Iteration 4 and the results from the user testing can be found in Section 5.2.

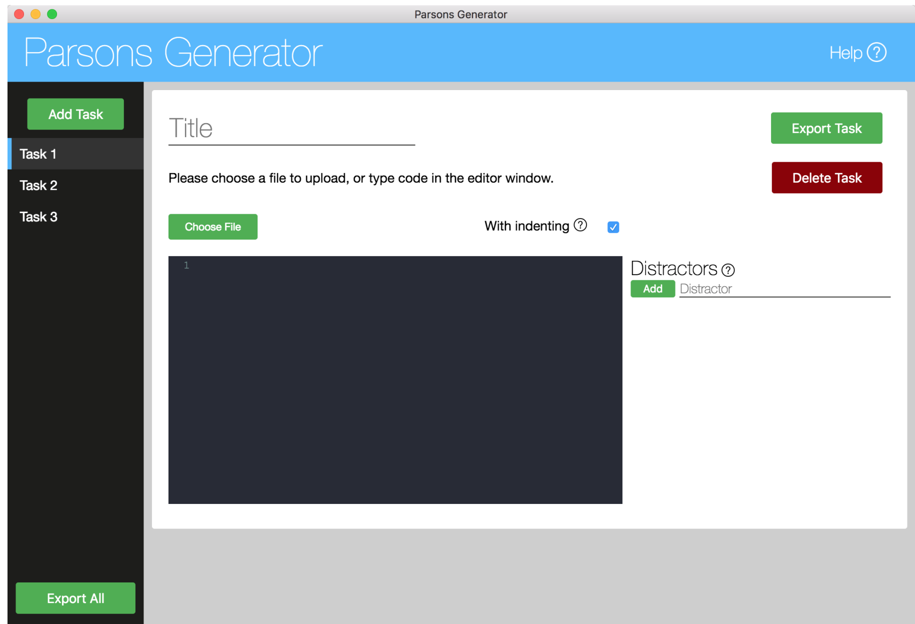

Figure 5.12: Application after Iteration 4

Testing

As this iteration focused on preparing the IT artefact for user testing, and only two major features were implemented, the need for extensive addition of unit tests was not needed. These features, distractor additions and helper text, were tested by doing use cases, and adding unit tests to the QTI Converter.

In addition to the unit and system testing stage, the user testing stage was performed during this iteration. This stage is discussed in Section 5.2.

Use cases can be found in Appendix C.4.

Evaluation and feedback

Since the main focus of this iteration was to prepare for and perform user testing, not too much development was conducted. The user tests generated a substantial amount of feedback, which is discussed in Section 5.2. The main points and feedback discovered during the testing can also be found in Table D.4. This feedback was further discussed and examined together with the thesis supervisor to plan how to proceed. Taking into consideration the time left, and apart from fixing minor bugs and improvements, it was decided to focus on and implement four major features during the last iteration. These features were permutations, task description, preview, and the generation of multiple tasks using the same algorithm, and these will be discussed further in the next iteration. With these remaining features, the IT artefact would be ready for real-world use.

To conclude the iteration, the developers felt that the development done was satisfactory and that the feedback gathered from the user tests was extremely informative and helpful.

5.1.6 Iteration 5

Goals

Functional Requirements for Iteration 5
(Note that these are the new requirement added for this iteration. Previous requirements were still relevant)

ID	Title	Description
9	Generate multiple unique tasks (w/ different distractors)	The IT artefact should be able to use a subset of the given distractors to generate multiple unique tasks (of the same algorithm) with different distractors
12	Task description in English, Norwegian and Nynorsk	The IT artefact should support the creation of a task description in English, Norwegian and Nynorsk. (This can be done in Inspera itself rather than in the IT artefact, but might be problematic if many tasks are to be generated)
13	Preview	The IT artefact should show the user a preview of the task to be generated so potential errors are discovered as early as possible (before exporting and uploading the task to Inspera)

19	Permutations	The IT artefact should support code permutations to avoid incorrect grading if students find other correct rearrangements of the code. The generated task must make sure that all correct permutations are properly represented. The IT artefact should also provide proper warnings if there exists any permutations that count as false positives

Tasks for Iteration 5

Title	Definition of done
Add preview	Create a preview in the GUI with live updating
Improve help-functions	Add additional information to the help popups that the preview does not improve understanding for
Fix file-chooser bugs	1) Make the file chooser a modal, to force the end-user to make an action before moving on 2) When exporting, a check should be carried out to make sure the filename doesn't already exist. If it does, ask the end-user if it should be overwritten.
Improve width accuracy of generated drag areas	Find exact font-metrics to accurately calculate width of each drag area when generating them
Move 'Delete task' button	Move the 'Delete task' button from the top right corner to the tab system (task overview). The delete button should appear as a cross button when hovering over a task. Add a confirmation box before deleting the task as well
Add task description	Add task description with tabs for Norwegian, English and Nynorsk.
Convert HTML text to XHTML	The task description will be received as HTML, and must be converted to XHTML for Inspera
Enable/disable permutations functionality	Make it possible to enable and disable permutations functionality. Provide warning message that permutations cannot be implemented in a 2D Parsons problem
DAG GUI	Create a user interface that makes it possible to easily create a DAG indicating precedence of code lines
Permutations preview	Make a preview window for the generated permutations

False positive permutations indication	Clearly indicate what permutations are false positives
Implement topological sort	Implement a topological sort function that can receive a DAG and return all topological sorts
Implement transitive closure	Implement a transitive closure function that can receive a matrix and find all transitive closures
Find all Inspera permutations based on transitive closure	Create a function that uses the transitive closure to find all permutations Inspera will deem completely correct
Calculate distractor subsets	The front-end has to calculate the number of possible subsets that can be created based on the given number of distractors the user wants included in each task (This has to be the max number of tasks possible to generate)
Implement distractor subsets	The back-end not only has to calculate the number of subsets, but also find them all and add them to the specific tasks the user wants to generate
Make identical drag areas connected to same drop areas	All identical drag areas has to be connected to the same drop areas in the generated task
Minor fixes	- Move 'Add' button for distractors behind the input-field - Write 'Enter text here...' to clarify input fields - Move 'Export task' button to the bottom right to better indicate its the last step to perform - Fix the copy paste bug (copy pasting worked in development builds, but was forgotten to implement in release builds) - Fix grammatical errors ('Indentation', not 'Indenting')

The goal of the final iteration was mainly to deal with the feedback gathered from the experiment, but also to implement the final necessary requirements. These requirements were chosen to make sure the final release of the IT artefact had the most important features implemented and to make sure it was ready for real-world use. Together with the supervisor, it was established that the main requirement that was missing during Iteration 4, was the ability to generate multiple tasks from a single algorithm, using subsets of distractors. As mentioned in Section 2.1.2, being able to generate multiple tasks can work as an effective tool against cheating, and

can also help support formative assessment. And if the IT artefact was to support the generation of multiple tasks, it would also require the implementation of a task description. Previously, the IT artefact only generated a single task, and a user could then just as easily write the task description in Inspera, but this was no longer an option. The addition of permutation support was also highly desired by the thesis supervisor since manually finding and adding permutations could quickly become a difficult and tedious task.

After iteration four, a lot of useful feedback was gathered from the user testing. Even though the users seemed satisfied with the IT artefact, each test participant had something to share in regards of possible improvements on the IT artefact. The feedback from all of these individuals accumulated to a long list of functionalities to be implemented, adjusted, or removed.

Implementation

The first addition to the IT artefact, was the task description box, which supported three languages. Users felt that having to do additional work in Inspera to complete the task was unnecessary and annoying, especially if one were to generate multiple tasks. The user wanted the possibility to create an entire task using the IT artefact, which included adding a description. The task description was implemented using NicEdit[3], which is a JavaScript inline content editor.

One of the guidelines for uploading your own QTI to Inspera, is to make sure Inspera supports the features you upload. In regards to the task description, Inspera had no support for changing font type, font size or color, and this meant that the IT artefact could not support these features either. Because if one uploads a highly formatted text, Inspera will not remove the unsupported formatting. And it is uncertain how the unsupported formatting will eventually affect the uploaded tasks, which might pose additional risks for exam questions. So to minimise this risk, the IT artefact task description should try to only support the same formatting and features as the Inspera task description does. In Figure 2.14 one can see Insperas task description toolbar with all its supported features. By using NicEdit, it was possible to disable the unwanted formatting features, but it was still possible for a user to copy and paste in formatted text to the editor. If a user were to do this, the IT artefact does not provide impenetrable validation of the given text, which means that the formatted text might be uploaded to Inspera. As a side note, the given HTML must be converted to XHTML to be properly placed in the XML files. And the IT artefact would give an error message if it found out that the given HTML had too much formatting to be converted to XHTML, but did not provide

[3]http://nicedit.com/

any additional HTML validation. This was one of the drawbacks with the implementation, but further validation of the given HTML was seen as future work and is also mentioned in Section 7.2.

An explanation of, and motivation behind the support for permutations, the different implementation options and the final implementation of permutations are all discussed in Section 5.1.7.

A preview of the final task was added to show the user how the task would end up in Inspera. The reason for this was two-fold, (1) to highlight the functionality of the 2D Parsons toggle, and (2) give a better picture of what kind of task the IT artefact was creating. The user feedback revealed that users did not understand what the 2D Parsons toggle did or meant. The easiest way to accommodate this problem, was to explicitly show through a preview what the toggle does, giving the user immediate feedback. The feedback also revealed that users not always understood the connection between Inspera and the IT artefact, making the entire task creation process a bit confusing. By adding the preview, users might be more aware of what is being created.

During this iteration, task settings were also added to the IT artefact, where number of tasks to create, number of distractors to include, and a 2D Parsons toggle was options to be decided. Having added a number of distractors, and deciding to include only a subset of those distractors, the IT artefact was able to create a number of unique tasks. The upper limit of unique tasks was decided by the binomial coefficient. The settings area was placed beside the preview to highlight the effects of each setting. Toggling the 2D Parsons changed the drop areas in the preview from a single column to multi column grid, as it would in the task to generate. Adding more or less distractors to be included in the task also changes the number of distractors in the preview. Distractors in the preview were masked (named 'Random Distractor') to differentiate them from the correct code lines and to highlight that a random subset of the distractors would be added to each generated task.

Some additional bug fixes and minor changes were made to the IT artefact as well. For instance, because of the program sequence (glance sequence), the first action of the application should be placed in the top left corner, and the last action in the bottom right, as users naturally wants to start top left and work his/hers way down right (Kurosu and Kashimura, 1995). For this reason, the 'Export Task' button was moved to the bottom right corner, as this is the last performed action in the application.
The rest of these minor fixes can be seen in the Task Overview for Iteration 5 and do not require additional discussion here. Figure 5.14 shows the final flowchart of the QTI Converter, while Figure 5.13 shows the final GUI of the IT artefact.

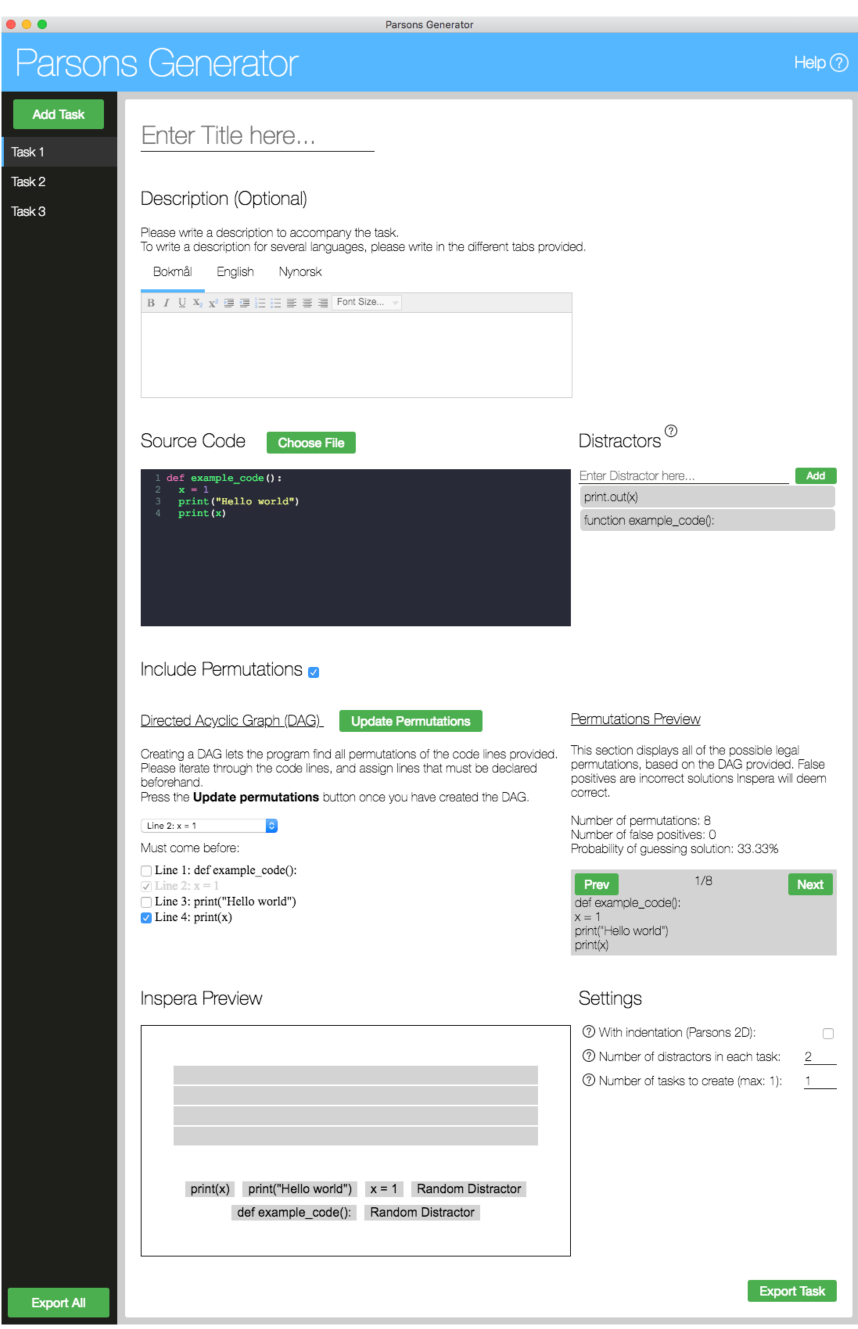

Figure 5.13: Application after Iteration 5

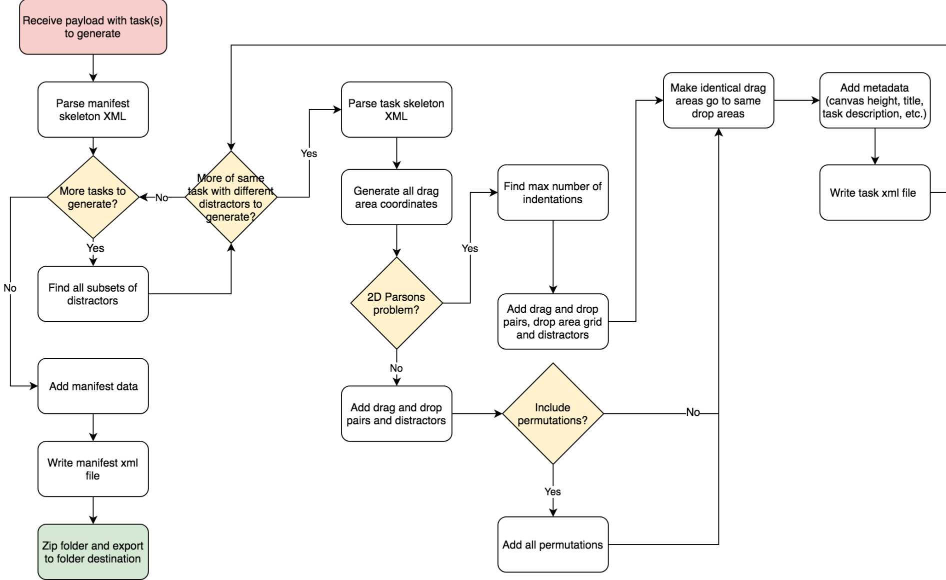

Figure 5.14: Flowchart of QTI Converter in Iteration 5

Testing

There were many features added during this iteration, which resulted in many new test cases. The user interface, with its additional functionality, was substantially tested with use cases, and the back-end was tested with both unit tests and use cases. Even though not every code path had its own unit test, every function in the QTI Converter had one or more unit tests.

To fully unit test all functions in the back-end code, two separate testing stages had to be done. The first stage, as seen in Figure 5.15, was done before run-time, while the second stage, as seen in Figure 5.16, was done during run-time. The reason for this was that the part of the back-end dealing with XML parsing required a running browser to be tested, which the developers were unable to do with the testing framework used. The solution was to manually write tests asserting in run-time, and logging to the console, for unit tests regarding XML parsing, while the rest of the back-end was tested with the testing framework Mocha[4].

Use cases can be found in Appendix C.5.

[4]https://mochajs.org/

```
> ParsonsGenerator@1.0.0 test /Users/jwjorgen/Documents/MasterThesis/masteroppgave-js/src
> mocha



  ✓ Test CalculatePermutations class on medium matrix
  ✓ Test CalculatePermutations class on small matrix
  ✓ Test CalculatePermutations class with cycle error 1
  ✓ Test CalculatePermutations class with cycle error 2
  ✓ Test GraphClosure on small matrix
  ✓ Test GraphClosure class on medium matrix
  ✓ Test GraphClosure class on large matrix
  ✓ Test GraphClosure class on medium matrix reaching all
  ✓ Test GraphClosure class and CountTransitiveClosurePossibilities class on small matrix
  ✓ Test GraphClosure class and CountTransitiveClosurePossibilities class on medium matrix
  ✓ Test functions in CountTransitiveClosurePossibilities class
  ✓ Test TopologicalSort class
  ✓ Test TopologicalSort class when reaching cutoff (Error) (61ms)
  ✓ Test TopologicalSort class
  ✓ Test random numbers
  ✓ Test generating ID
  ✓ Test find tab size
  ✓ Test getting max tabs
  ✓ Test counting tabs in one line
  ✓ Test stripping string
  ✓ Test canvas height
  ✓ Test get random subset of distractors

  22 passing (95ms)
```

Figure 5.15: Unit tests for QTI Converter

Figure 5.16: Run-time unit tests

Evaluation and feedback

As this was the last iteration, and after the user testing phase, the only feedback received was from the supervisor. The final version of the prototype was shown to him, and he was satisfied with the end result.

To conclude all of the iterations, the developers were satisfied with the final outcome of the prototype. A few major features were implemented in the last iteration, all of which achieved a satisfactory level of done. As this was a prototype, there were additional features and ideas that could be potentially implemented, all of which are discussed in Section 7.2 (Future Work).

5.1.7 Permutations

Introduction and Motivation

Permutations are in this context defined as the rearranging or reordering of code lines. When the IT artefact is given a code snippet by the user, there might exist rearrangements of the given code lines that result in different, but completely correct and legitimate code snippets. Figure 5.17, 5.18, and 5.19 shows a permutation of a short code example and how this kind of permutations must be represented in Inspera.

Although this example is extremely easy, it shows how some code snippets might have rearrangements of code lines, which result in the same wanted functionality as the original code. From the code example in Figure 5.17 there exists a total of $3! = 6$ code line permutations, where Figure 5.17 and 5.18 show the only two permutations that results in the wanted functionality, from this point on referred to as 'correct permutations'. So the correct permutations are rearrangements of the

```
1    x = 1
2    y = 2
3    print(x+y)
```

Figure 5.17: Code example

```
1    y = 2
2    x = 1
3    print(x+y)
```

Figure 5.18: Permutation

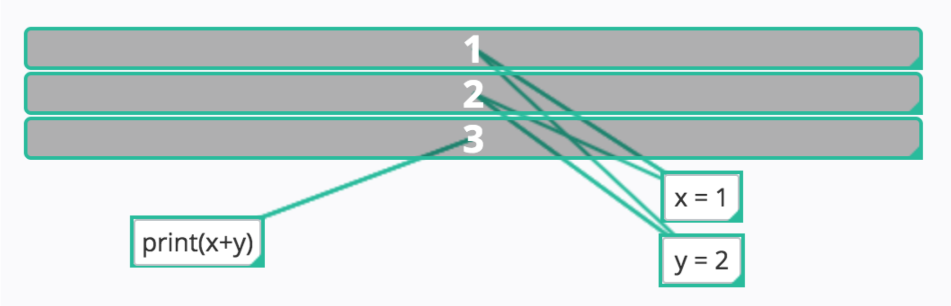

Figure 5.19: Permutations represented in Inspera

code that result in equivalent behavior.

When creating exam questions for Inspera, it is extremely important to find and take these correct permutations into account to avoid falsely failing students that happen to find one of the correct permutations. In larger algorithms, manually findings these permutations can be quite difficult and time-consuming, and being able to automatically find these permutations could be of great value. And since not supporting correct permutations of an algorithm when generating exam question can have serious consequences, this feature was decided to be of great importance to the IT artefact. Together with the quantitative data gathered from the user testing, this feature was added to the list of improvements, as seen in Appendix D Table D.4.

As one finds all the correct permutations of a code snippet and implements these permutations in Inspera, one also runs the risk of creating answers that are 'false positives'. Since the tasks created in Inspera has no conditional logic or control of the preceding or following code lines, all logic regarding the permutations has to be represented simply by having a drag area go to multiple drop areas. In Figure 5.19 one can see that the *'x = 1'*-drag area can go to drop area 1 and 2. In more advanced algorithms, these simple connections between a drag area and multiple drop areas can allow for incorrect permutations that Inspera will deem completely

correct.

In the following example, Figure 5.20 shows all correct permutations of a new code example. These 5 code lines have a total of $5! = 120$ permutations, where this figure show the six correct permutations that result in the exact functionality we want.

```
1 x = 1
2 y = 2
3 z = x*2
4 print(y)
5 print(z)
```

```
1 x = 1
2 y = 2
3 print(y)
4 z = x*2
5 print(z)
```

```
1 x = 1
2 z = x*2
3 y = 2
4 print(y)
5 print(z)
```

```
1 y = 2
2 x = 1
3 z = x*2
4 print(y)
5 print(z)
```

```
1 y = 2
2 x = 1
3 print(y)
4 z = x*2
5 print(z)
```

```
1 y = 2
2 print(y)
3 x = 1
4 z = x*2
5 print(z)
```

Figure 5.20: All correct permutations of the given code example

With these correct permutations, the following task has to be created in Inspera, as shown in Figure 5.21. Here one can see all the connections between the given drag and drop areas.

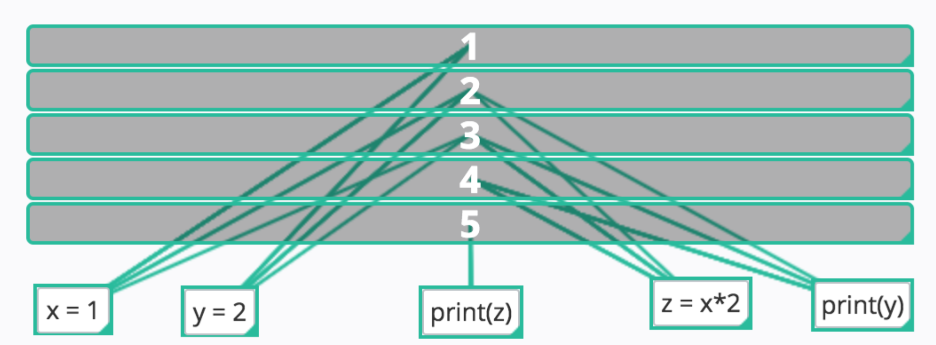

Figure 5.21: Permutations represented in Inspera

But it is from these 'simple' connections the possibility for false positives also arises. For example, one can see that the code line *print(y)* can be placed at line number 2, but it can only be placed here correctly if the code line *y = 2* is at line number 1. But this kind of conditional logic cannot be represented with these connections, which means that the following permutations shown in Figure 5.22 will be deemed correct by Inspera, while these in reality are incorrect permutations. These permutations are the false positives.

The false positive permutations shown in Figure 5.22 show the combinations that will give a student full score on the Inspera task, but there will also exist other rearrangements or incomplete answers that wont necessarily give full score, but still give more points than it should.

```
1   x = 1
2   print(y)
3   y = 2
4   z = x*2
5   print(z)
```

```
1   y = 2
2   z = x*2
3   x = 1
4   print(y)
5   print(z)
```

Figure 5.22: All false positive permutations of the given code example

As one can see here, knowing all the correct permutations and knowing if there exists any false positive permutations can be extremely important, at least when creating exam questions.

Implementation

To properly support permutations and finding false positives, the IT artefact should preferably discover these automatically, simply by being given the initial code. This would require some logic to figure out and find all correct permutations. As 8 lines of code would generate $8! = 40320$ permutations, one possibility is to have each of these permutations go through a compiler to see if the given arrangement still give the same results. Trying to find, create, and use one or multiple compilers to support all kinds of code snippets, faulty permutations, and inputs in multiple programming languages is an incredible comprehensive and extensive thing to do, and would most likely require a lot of time, and result in a flawed and incomplete solution given the time-frame of this thesis. Trying to use a compiler to find the correct permutations was therefore discarded.

The main goal was then to find a way to generate all the permutations without having to understand the code. The next idea was to treat this problem as an ordering problem, where the solution would be to find all possible orders given some constraints to keep the code snippet correct. By using constraints or rules between each line of code, one could end up with a partial order that could generate all possible total orders. And this could be obtained by using topological sorting. Topological sorting is also known as a restricted permutation problem (Atkinson, 1999), which focuses on finding permutations that are consistent with the given set of restrictions. Topological sorting requires a directed acyclic graph (DAG) indicating precedences among events, or in this case, lines of code, to create a linear ordering where all its directed edges uv, u comes before v in the ordering (Cormen et al., 2009). Kalvin & Varol (Kalvin and Varol, 1983) compares different algorithms for finding all topological sortings, and by implementing the Knuth–Szwarcfiter Algorithm (Knuth and Szwarcfiter, 1974) one is able to generate all topological sorting arrangements with a time complexity of $O(m + n)$ for each solution generated. Note that without any edges in the DAG, the worst-case time

complexity becomes $O(n!)$.

A DAG showing all precedences among lines of code means that a directed edge between *y = 2* and *print(y)* is equal to stating that *y = 2* must come before *print(y)* in the code. If a complete DAG is given to a topological sorting algorithm, one would be able to generate all topological sorts, which in other terms are all the correct permutations of the code.

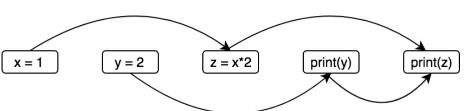

Figure 5.23: DAG indicating precedences among lines of code

Given the little time that was left of this project, having the user manually create the DAG was seen as the best solution. An advantage of this is that a DAG can create all possible topological sorts, so that further analysis of the code itself is not necessary to find all correct permutations. The disadvantage is that this manual process can be time-consuming and prone to user errors.
A better solution, might have been to fully, or partially, automate this process. To achieve this, one could for example do a syntax analysis of the code, given some predetermined rules such as variables appearing on the right hand side of a '=' must be preceded by a declaration of said variable.

To manually create a DAG, the interface shown in Figure 5.24 was implemented in the IT artefact. The main idea here was to first choose a line number from the top drop-down menu, followed by checking off which lines of code that has to come after it. So Figure 5.24 shows how the top left edge in Figure 5.23 is created in the IT artefact by the user. By filling out which lines must precede every other line, the IT artefact can now automatically generate all correct permutations simply by running topological sort on the given edges.

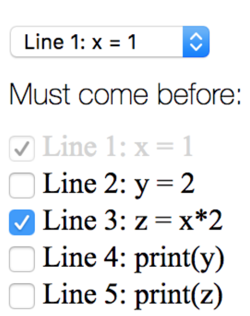

Figure 5.24: Interface for creating a DAG

While the correct permutations can now be found, it still remains to find all false positive permutations that will be generated due to the 'simple' connections Inspera creates. Here one can simulate the logic created in Inspera by finding the transitive closure of the given DAG. A transitive closure algorithm takes a directed graph given in the form of an adjacency matrix and determines whether the graph contains a path from vertex i to j or not (Cormen et al., 2009). By running a transitive closure algorithm, the

returned adjacency matrix shows all reachable vertices from the given vertex. Figure 5.25 shows the corresponding adjacency matrix given the DAG from Figure 5.23. The transitive closure algorithm simply adds new edges from vertices to any additional transitively reachable vertices.

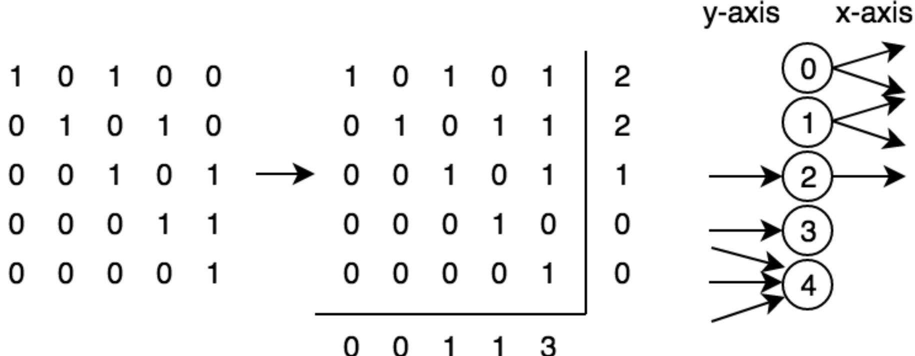

Figure 5.25: Transitive closure

Looking at row 0 in Figure 5.25, which represents code line 0 (x = *1*), the transitive closure of the given adjacency matrix shows that vertex 0 (code line 0) can reach vertex 2 and 4. With each row in the new adjacency matrix one can find the number of outgoing edges from that particular vertex by calculating the sum minus 1. The sum of each column minus 1 also shows the number of incoming edges to that particular vertex. And the interesting property of this new adjacency matrix, is that the number of outgoing edges correspond to the number of vertices that must come after the given vertex in all its rearrangements. Similarly, the number of incoming edges correspond to the number of vertices that must come before the given vertex in all its rearrangements. Figure 5.26 show how a vertex with one incoming edge and two outgoing edges in a code example of 5 lines will find its correct positions.

In broader terms, this ’Code line’-vertex in Figure 5.26 says that it can be placed wherever, as long as there is at at least one vertex preceding it and at least two vertices after it. This is the exact same logic that Inspera will follow with its connections. In Inspera this ’Code line’-vertex will be connected to the green drop areas (2 and 3). So by using the transitive closure and finding the incoming and outgoing edges of all vertices, one knows which drop areas each line of code will be connected to, which in Figure 5.26 are marked green. Thus, by using the transitive closure algorithm to find all connections, we properly simulate how Inspera will treat the task.

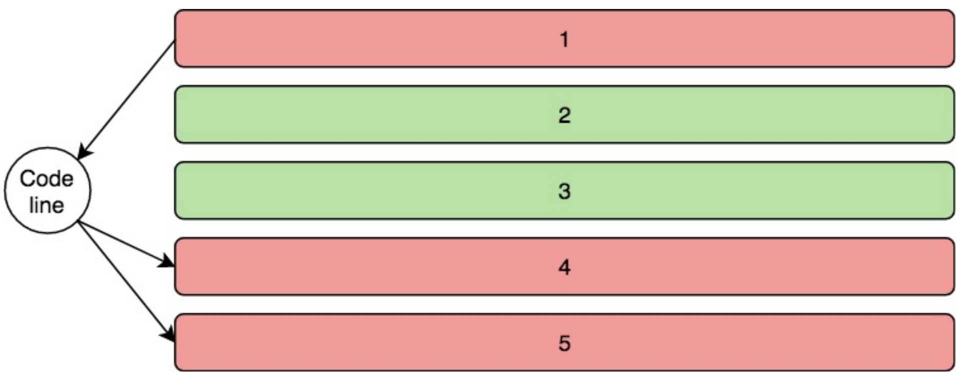

Figure 5.26: Code line vertex with its preceding and following drop area positions. The code line is thus connected to the green drop areas in Inspera.

Since we now know all the placements Inspera will deem correct for each line of code, the next step is to find all possible permutations of the algorithm given these connections. Looking back at Figure 5.25 one can see that vertex 0 and 1 has two outgoing edges, which means they can be placed wherever, except at the two bottom lines. By running a recursive algorithm, one can find all possible permutations that follows the rules given by the transitive closure algorithm. One then ends up with a set of permuations that all will be deemed completely correct by Inspera. If we call the the permutation set created by the topological sorting, set A, and the permutation set created by the transitive closure, set U, the set difference of U and A (that is, $U \setminus A$) are the false positive permutations the task will contain.

The number of correct permutations and the number of false positive permutations are good indications of the quality of the task itself as well. If a task has many correct and false positive permutations, one should probably re-evaluate the task in its entirety. In this case, a user could choose to properly address the issue by changing the code, choosing another task, or simply discarding it. Section 7.2 (Future Work) also discusses some other possibilities and features that could help users reduce the number of permutations created by a task, but that has not been implemented yet.

Since the calculation of these permutations can be quite demanding with larger algorithms, both the topological sorting (which has a worst-case time complexity of $O(n!)$) and the transitive closure algorithm (which has a worst-case time complexity of $O(n^3)$) was implemented with kill-switches that stops the algorithms at a certain cutoff. If the DAG has no edges, the topological sort will have to find $n!$ permutations where n is the number of vertices. To avoid this kind of scaling, the

cutoff was set to 1000 permutations.

The next issue was to find a way to properly communicate these permutations with the user of the IT artefact so one could clearly see what correct permutations would be generated, as well as all the false positives. Giving as much information to the user as possible is extremely important, such that nothing unexpected is generated and added to a task for an exam. Figure 5.27 shows the final implementation of the permutation support in the IT artefact. By first selecting that one wants to include permutations in the task, one then has to fill out the DAG and click the 'Update Permutations' button. All the correct permutations and false positives, as discussed above, are then found and displayed in the Permutations Preview section. This preview lets the user go through all the permutations as well as the false positives. This kind of preview is also necessary to double check if there exists any correct permutations that should not be correct, which might come from a lack of information in the DAG. As discussed in Section 2.6, proper feedback about the automation states, actions and intentions must be provided to give the user good situational awareness (Parasuraman and Riley, 1997)(Parasuraman et al., 2000).

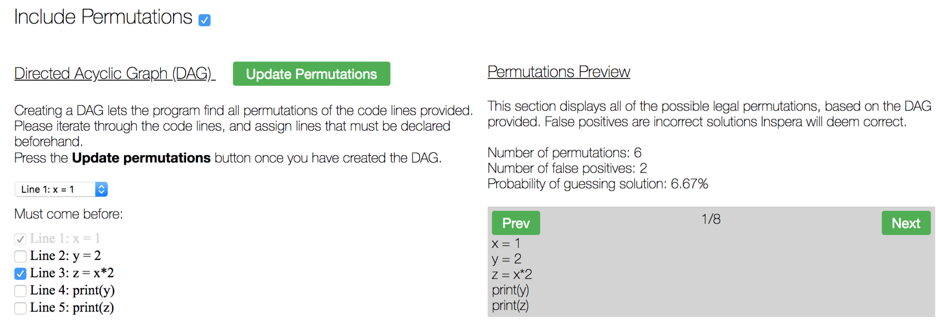

Figure 5.27: Final implementation of permutations in the IT artefact

Limitations

As previously mentioned, one of the limitations of this implementation is that the user has to spend some time manually creating the DAG, but since this input makes it possible to properly find all permutations, it was considered to be a worthy investment and use of time.

```
1  for i in range(10):
2      for j in range(5):
3          print(i+j)
4  x = 3
5  print(x)
```

Figure 5.28: Code example

The greatest limitation of this implementation was that the researchers were unable to accurately find and add the logic required to make the permutation support work for 2D Parsons problems. A lot of time was spent trying to find rules that could help the algorithm understand where new indentation levels would be needed and where one can remove them, but no global rules was found. One example of a more difficult case to create indentation rules for can be seen in Figure 5.28. Here, one can see that line number 4 ($x = 3$) simply has to be declared before *print(x)*, but with Python, it is completely correct to give line number 4 both one and two additional indentation levels. These additional indentations would of course lead to a worse solution since the same variable would be declared multiple times, but this example illustrates the complications that might occur when supporting permutations with indentation. Finding and creating logic to support these kinds of legal, but different, indentation levels is largely dependent on the programming language as well, since different languages have different rules of where and how things must be declared. Since no proper logic for handling indentation levels was found, permutations was not implemented for 2D Parsons problems. If users create a 2D Parsons problem, they can still create a DAG and see the permutation previews shown in Figure 5.27 to evaluate the quality of the task. But the IT artefact will also display a warning message letting the users know that these permutations will not be implemented in the task generated.

```
1  public static void main(String args[])
2  {
3      for (int x = 2; x <= 4; x++)
4      {
5          for (int y = 2; y <= 4; y++)
6          {
7              System.out.println(x+y);
8          }
9      }
10 }
```

Figure 5.29: Java code example

Another limitation was that the permutations did not account for lines of code that were similar, for example end-brackets in Java (*}*). Therefore, when calculating the number of correct and false positive permutations in the GUI, the IT artefact might show two or more permutations that look completely identical to each other, but in reality is not since the identical lines are rearranged. To make sure this limitation only affected the GUI, and not the actual generation of the QTI, additional logic was implemented to make sure that all similar drag areas were connected to the same drop areas in the generated task. This logic is extremely important, at least for programming languages such as Java, where there might exist multiple end-brackets, as seen in Figure 5.29. Since identical lines in a drag and drop task is impossible to distinguish from each other, one must make sure all identical lines can be placed at the same positions. In the case of Figure 5.29, one must make sure that all starting-brackets can be placed on the same positions, and that all the end-brackets can be placed on the same positions.

5.1.8 Technology Choices

Deciding on which technologies to use is a cumbersome process. There are pros and cons for each framework, programming language, and technology which has to be assessed and thoroughly worked through. For this thesis, the technological topics discussed were web framework vs. desktop application, framework to use (if any), and programming language.

Web Application vs. Desktop Application

Complexity and Maintenance
The focus of this thesis was not centered around the application itself, but rather the effect of such an application. For this reason, creating an application with as little complexity as possible would be beneficial, as this would reduce development and maintenance time substantially when creating prototypes and proof of concepts.

Web applications usually have a more divisive architecture than desktop applications, making the border between front-end and back-end more clear. This helps reduce the overall complexity of the application, but introduces the need for communication between server and client.
Desktop applications on the other hand, introduces the problem of keeping an updated and maintained version of the application at all times. With a web application, the latest code is pushed to the server, and all clients have access. A desktop application requires the user to download the newest version each time a revision is done, which is not necessarily in the interest of a user.

Not having the latest specs is a problem that might occur in both instances. A web application can be restricted by browser versions, and a desktop application might require the user to download the latest version of tools like Java. The operative system is also something that has to be taken into consideration, as the application has to be functional on Microsoft Windows[5] and MacOS[6].

Internet Access
Having the application connected to the world wide web gives arise to several advantages such as immediate updates (as discussed in previous section), possibilities of enhanced machine learning, and easier access. A web framework is, of course, connected to the web, which makes internet access a requirement. Today, this should not be problematic, but is still something that has to be taken into consideration. A desktop application might also be connected to the web, but this is

[5]https://www.microsoft.com/en-us/windows
[6]https://www.apple.com/lae/mac/

not a requirement, making offline use possible.
One advantage of having the application connected to the web is the possibility of machine learning on a greater scale. The application can collect information from multiple sources (application instances), and thus gather more training data for machine learning.

Performance and Design

Design wise, there are libraries for both web applications and desktop applications that are undoubtedly good enough for our purposes. Today, there is a wide variety of modules and extensions that allows for web frameworks to operate on desktop applications, and thus, deciding on the correct design libraries is not of importance. Earlier, one had to take into consideration which operating system that the application had to support, but with the extensions that exist today (e.g. Electron[7]), this is not something that has to be taken into consideration.

Performance wise, the desktop application has the advantage of not relying on network speed. The application back-end is integrated directly with the front-end, and the payloads between them are transferred with negligible delay. A web application is reliant on the internet connection, and thus, poor or no connectivity can create problems and/or user dissatisfaction.

Another advantage of the desktop application is the possibility of local persistence, making it possible to exit the application and then resume at the same instance later on. This is of course possible on a web application as well, but requires the addition of a login system, which increases the complexity of the application.

Cost

A web application requires hosting on a server, and a domain name. A desktop application on the other hand, has no hosting cost.
A potential database setup would require additional hosting expenses.

Security

With web applications, a security breach is more likely to happen than on an offline desktop application. When payloads are sent over the web, the possibility of package sniffers are existent. And since the data contains information regarding exams, security is of utmost importance. In addition, a web framework might make use of Single Sign On (SSO) connections and online storing of information in databases, making security breaches catastrophic.

[7]https://electronjs.org/

In an offline desktop application, the most imminent security risk is someone having access to the computer, which is not something that this application can prevent. The desktop application can be equipped with a login system, but this would not be a priority.

Summary of Web vs. Desktop Application

	Web Application	Desktop Application
Complexity and Maintenance	**Pros:** - Live updates **Cons:** - Must be compatible with all browsers	**Pros:** - No need for communication with a server **Cons:** - New versions must be downloaded - Need versions for all operating systems
Internet Access	**Pros:** - Possibilities for easier accessibility - Possibilities for enhanced machine learning **Cons:** - Require internet connection	**Pros:** - Does not require internet connection **Cons:**
Performance and design	**Pros:** - Good support for libraries **Cons:** - Relies on network connection - Persistence requires added complexity	**Pros:** - Good support for libraries - Negligible transfer time of payloads - Possibility of local persistence **Cons:**
Cost	**Pros:** **Cons:** - Requires a server for hosting - Requires a domain name - A database would require hosting	**Pros:** - Requires no hosting expenses **Cons:**
Security	**Pros:** **Cons:** - Security breach is a potential risk - Package sniffing is a threat	**Pros:** - Nothing is sent/stored online **Cons:**

Table 5.6: Web vs Desktop Application

Framework

Java with JavaFX
Java[8] and JavaFX[9] are both widely used in application development because of their robustness and soundness. But these attributes attracts more complex code, which is not ideal for an application striving to be as easily developed and maintained as possible. Java code can be quite verbose, making the simplest of functions unnecessary long, compared to e.g. Python.

JavaFX was released in 2008, and even though it looks modern and clean, it does not really provide a native look and feel on a desktop application.

Python with Tkinter and Flask
Python[10] shines where Java does not; The language is succinct and compact, advocating for smaller applications. And Python supports an enormous library of modules, rendering a developers life easier. But even with such a wast library, there are not really any good modules for creating desktop user interfaces. One of the contenders is Tkinter[11], which does not have a large active user community and have not really been updated in recent years, making it difficult to work with, compared to other user interface tools.

If one were to split up an application, using Python for back-end, and some other framework/language for the front-end, there is a necessity of opening a port for communication between the two parts. For this, Flask[12] is an excellent Python module. It is a lightweight web framework with minimal amount of dependencies on external libraries and it can thus operate as an API for the back-end. The drawback of this, is that the (desktop) application has to pack the Flask module into the release build, and then on application start, a port has to be opened in the background to allow for communication between front- and back-end. This would cause unnecessary complexion for a simple application.

Electron with JavaScript
Electron[13] is a software framework developed and maintained by GitHub[14], and

[8] https://www.oracle.com/java/
[9] https://wiki.openjdk.java.net/display/OpenJFX/Main
[10] https://www.python.org/
[11] https://wiki.python.org/moin/TkInter
[12] http://flask.pocoo.org/
[13] https://electronjs.org/
[14] https://github.com/

is used for development of desktop applications. It utilizes Node.js[15](Tilkov and Vinoski, 2010) for back-end and Googles[16] Chromium[17] for front-end. As a result, the entire application can be written in JavaScript, which excels in both front-end and back-end.
Through NPM packages[18], Electron supports deployment to all the major operating systems, which makes it easier to create and deploy an application quickly in an iterative development process. Having the application not platform specific, makes it easier to test on a wide range of users and get fast feedback.

With the use of JavaScript, the support for web frameworks are also present. Electron allows for the integration of frameworks like Facebooks[19] React[20] and Googles Angular[21]. This would be beneficiary for a larger desktop application, but using a framework was initially considered overkill for this IT artefact. The consequences of not using a framework is further discussed in Section 6.1.

Summary of Java vs. Python vs. Electron

	Pros	**Cons**
Java with JavaFX	- Robust - Looks modern and clean - Enormous package library	- Verbose code - Not a native look and feel
Python with Tkinter and Flask	- Succinct and compact language - Enormous package library	- Tkinter is outdated - Difficult to package Flask
Electron with JavaScript	- Enormous package library - Supports web frameworks	

Table 5.7: Pros and cons of frameworks

Final Decision

One of the goals of the development process was to choose an environment with as little complexity as possible, and focus on the support of a fast and iterative process. For this reason, the researchers found that building a desktop application,

[15]https://nodejs.org/en/
[16]https://www.google.com/
[17]https://www.chromium.org/
[18]https://www.npmjs.com/
[19]https://www.facebook.com/
[20]https://reactjs.org/
[21]https://angular.io/

using Electron, would be the best course of action.

5.1.9 Design

During the entire design process, several design topics were taken into consideration, such as affordances, feedback, colors, and user input. To optimise the user experience, it was important to create a system that was efficient, intuitive, and aesthetic. And the design choices greatly affect those parameters. This section refers to the implementation of the design theory discussed in Section 2.7.

Visibility

Visibility refers to the exposure of functions and operations available. Accommodating visibility, the task page of the application had all functionalities displayed on one page, such that there were no need for hyperlinks to navigate around each task. All operations and functions were intentionally visible all the time, with the exception of permutations, so that users would not have to spend time locating desired features. The reason permutations were hidden when not selected, was to not confuse users whether this feature was selected or not.

Feedback

To satisfy the user flow, all actions made by the user were to have some kind of feedback. Buttons either added/removed elements from the user interface, or opened a file dialog. Hovering the question marks immediately displayed an information box. And toggles provided immediate updates in the user interface. All of these feedback elements were meant to help the user through the application flow, and make for a smooth experience.

The most considerable feedback provided by the application was the preview window, which was updated whenever a user made an action. This feature provided a real time feedback on every decision, and reflected how the task would be presented in Inspera.

Constraints

The application did not pose many constraints on the user interface, as most features were optional. One of two constraints implemented was the inability to add an empty distractor to the input field. The other constraint provided was in regards to permutations, where code lines were not selectable to depend on each other.

That is, if code line 1 was selected to 'Must come before' code line 2, the opposite selection was disabled.

Consistency

Consistency is important for a user friendly system, allowing for few surprises. Having all buttons colored green (with the exception of buttons with delete functionality), using the same design for all helper icons, and using the same font for all text provided a recognisable pattern easy to learn and understand.

In addition, consistent with the reading conventions of western cultures, the application was designed with the glance sequence from top left to bottom right in mind. The initial button, "Add Task", was placed in the top left corner, as this is usually the first place a user glances when opening an application. Moreover, the flow of creating tasks went from top to bottom, from left to right. And finally, signaling the end of the user sequence, at the bottom right corner the export button was placed.

Affordance

It is important that the perceived affordance of elements corresponds to their intended functions, as this will improve the efficiency of the design and ease of use (Lidwell et al., 2010).

To give buttons the perceived affordance of clicking, the cursor changed pointer to a hand when hovering a button, in addition to the button itself decreasing the saturation of its background color. This affords the notion that something will happen when it is clicked.
The preview, on the other hand, was meant to have no interaction at all. For this reason, there were no changes when the cursor hovered the preview, affording the notion that nothing would happen if it was clicked.

The need for an integrated development environment (IDE)-like coding window was seen as necessary to afford the writing of code in the source code editor. Normal text editors have the perceived affordance of writing normal text, while a source code editor needs the perceived affordance of writing source code.

Color

The color palette chosen for the IT artefact consisted mainly of four colors: light blue, green, red, and grey. Red and green are complementary, and green and light

blue are analogous. Buttons were made fully saturated red or green, as they were to grab the attention of the user to be something clickable. The background colors of the IT artefact consisted of white and different shades of grey, which are cool and neutral, and would not interfere with the user attention.

Considering people affected by color vision impairment, the IT artefact was designed with as few colors from the red and green spectrum as possible. But, due to cultural conventions, this was not always the case. Delete buttons are, by convention, red and the most suitable color for download buttons are green. To compensate for this, the buttons were annotated either by a descriptive text or an icon.

Confirmation

The most critical and irreversible function a user could do with the application was to delete a task. In such a case, the user would have to re-upload the code file, add all distractors again, configure permutation options (if any), and tune task settings. This would be a great annoyance, and therefore the application provides a confirmation message whenever a task is about to be deleted.

Another confirmation, was the popup window whenever a user tries to export the task(s) with a file name that already exists in the target directory. Instead of immediately overwriting the existing file, the application asks for a conformation on this action.

User Input

To avoid duplicate work, and decrease possibilities of erroneous input, application input was reused whenever possible. For example, having the user type code into the source code editor (or upload a file), the IT artefact extracted information such as code lines, number of code lines, and indentations from the source code editor for re-use.

Input fields throughout the application was designed to provide definitive borders, prompt text, and color indicators on hover. All of these attributes strengthened the input functionality, and made them more apparent.

Lastly, upon completion of data entries, the system provided explicit feedback. Whenever task title changed, the title updated in the list of tasks. When distractors were added, they appeared in the list of distractors. And when the export button was clicked, and the task was successfully exported, a confirmation box appeared.

5.2 Experiment

5.2.1 Quantitative Data

In this section an overview of the quantitative data gathered will be presented. This data was gathered from the experiment conducted on the real-life users of the IT artefact. As previously mentioned, the experiment consisted of observation of the user-tests, a SUS questionnaire, and an interview. Among these three data generation methods, the quantitative data resulting from the observation and the questionnaire will be presented here.

The experiment was conducted with five participants in total. The following tables show the raw quantitative data gathered from these experiments. Participant 1-4 all completed the experiment as planned by the researchers, while participant 5 did not complete the given tasks and canceled the Inspera part of the experiment in frustration. After spending 18 minutes on Task 1 in Inspera, and still only be half done, participant 5 asked if they could skip the remaining parts of Inspera and instead describe what they theoretically would do next. Due to this cancellation in the experiment, some of the gathered data from participant 5 will not be presented. This person still completed the given tasks with the IT artefact, but since the numbers can not be properly compared to the Inspera process, the data is defective. The SUS and the qualitative data gathered from participant 5 was still used in the analysis.

While the performed tasks are all presented in the same order in these tables, the order was randomized during the experiments, as discussed in Section 4.1.2. The data is also just presented in this section, and will be discussed further in Chapter 6.

Participant 1	Time spent (sec)	User-Errors	Inaccuracies	Questions	System-Errors
Task 1 Inspera	620	1	2	0	1
Task 2 Inspera	1055	2	0	1	1
Total Inspera	**1675**	**3**	**2**	**1**	**2**
Task 1 App	70	0	0	0	0
Task 2 App	201	0	0	0	2
Total App	**271**	**0**	**0**	**0**	**2**

Table 5.8: Participant 1 data points

Participant 2	Time spent (sec)	User-Errors	Inaccuracies	Questions	System-Errors
Task 1 Inspera	796	2	2	1	1
Task 2 Inspera	444	1	2	1	0
Total Inspera	**1240**	**3**	**4**	**2**	**1**
Task 1 App	111	0	0	0	1
Task 2 App	71	1	0	0	1
Total App	**182**	**1**	**0**	**0**	**2**

Table 5.9: Participant 2 data points

Participant 3	Time spent (sec)	User-Errors	Inaccuracies	Questions	System-Errors
Task 1 Inspera	893	1	4	5	0
Task 2 Inspera	627	2	3	0	1
Total Inspera	**1520**	**3**	**7**	**5**	**1**
Task 1 App	179	0	0	0	1
Task 2 App	175	1	0	0	0
Total App	**354**	**1**	**0**	**0**	**1**

Table 5.10: Participant 3 data points

Participant 4	Time spent (sec)	User-Errors	Inaccuracies	Questions	System-Errors
Task 1 Inspera	885	4	3	3	0
Task 2 Inspera	550	1	2	2	0
Total Inspera	**1435**	**5**	**5**	**5**	**0**
Task 1 App	189	0	0	1	1
Task 2 App	129	0	0	0	1
Total App	**318**	**0**	**0**	**1**	**2**

Table 5.11: Participant 4 data points

The following figures uses the same data as the tables above, but visualizes the mean data. Since the sample size was so small, the researchers saw no need for additional quantitative data analysis to visualize or represent further findings. Figure 5.30 shows the main data, and visualizes the differences in the two processes.

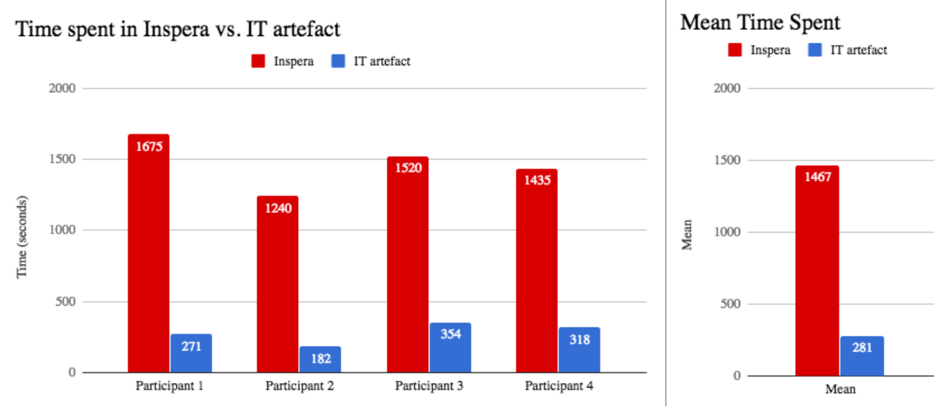

Figure 5.30: Visualization of time spent with Inspera and the IT artefact

With the 4 participants, the time spent in Inspera and time spent with the IT artefact are quite similar between the participants. Figure 5.30 shows every participants' total time spent in Inspera and total time spent in the IT artefact when solving the two tasks given in the experiment. The standard deviation of all time spent in Inspera was at 181 seconds and with the IT artefact it was 74 seconds. The most important aspect of this data is how in every case, the IT artefact was one fourth as time-consuming as Inspera. A much larger sample size and more in-depth data analysis would perhaps be necessary if the accumulated data had not shown such clear differences, but in this case, simply using common sense is enough to suggest that the IT artefact is more efficient. It is important to remember that none of the participants had any prior knowledge to the IT artefact, and most had not tried to manually create such a drag and drop task in Inspera either.

To better supplement the results regarding time spent creating the two tasks, the user performance time using the keystroke-level model (KLM) was calculated as well (Card et al., 1980). The actual calculation using the KLM and the results can be found in Appendix F, but the main results are visualised in Figure 5.31. Based on the Mean time spent from Figure 5.30, the ratio between time spent during user testing was exactly $1467/281 = 5.22$, while the ratio between time spent from the KLM calculation was $617/128 = 4.82$. These two ratios are fairly similar to each other, reducing the likelihood that the measurements from the user testing was generated by chance.

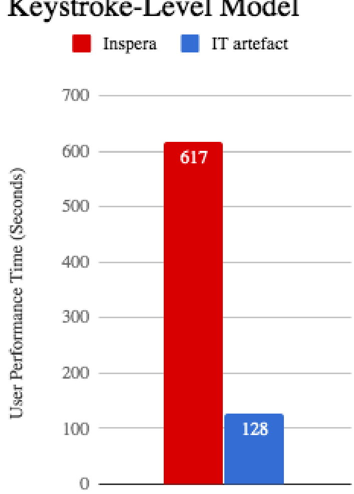

Figure 5.31: Calculated User Performance Time for both tasks using KLM

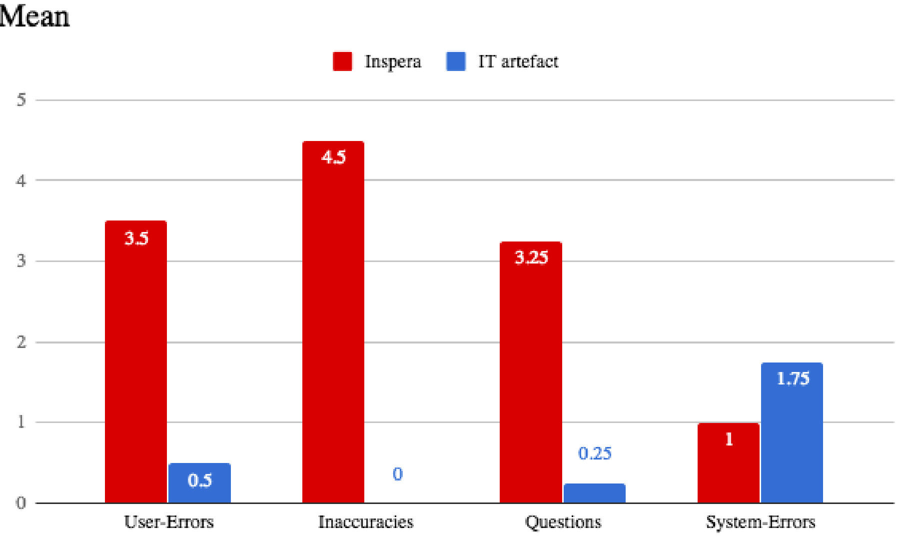

Figure 5.32: The mean of some gathered data points

Figure 5.32 shows the additional data points gathered during the user testing. These show that while the IT artefact still had some bugs and system-errors during the testing, the overall process proved to be more intuitive, with less questions

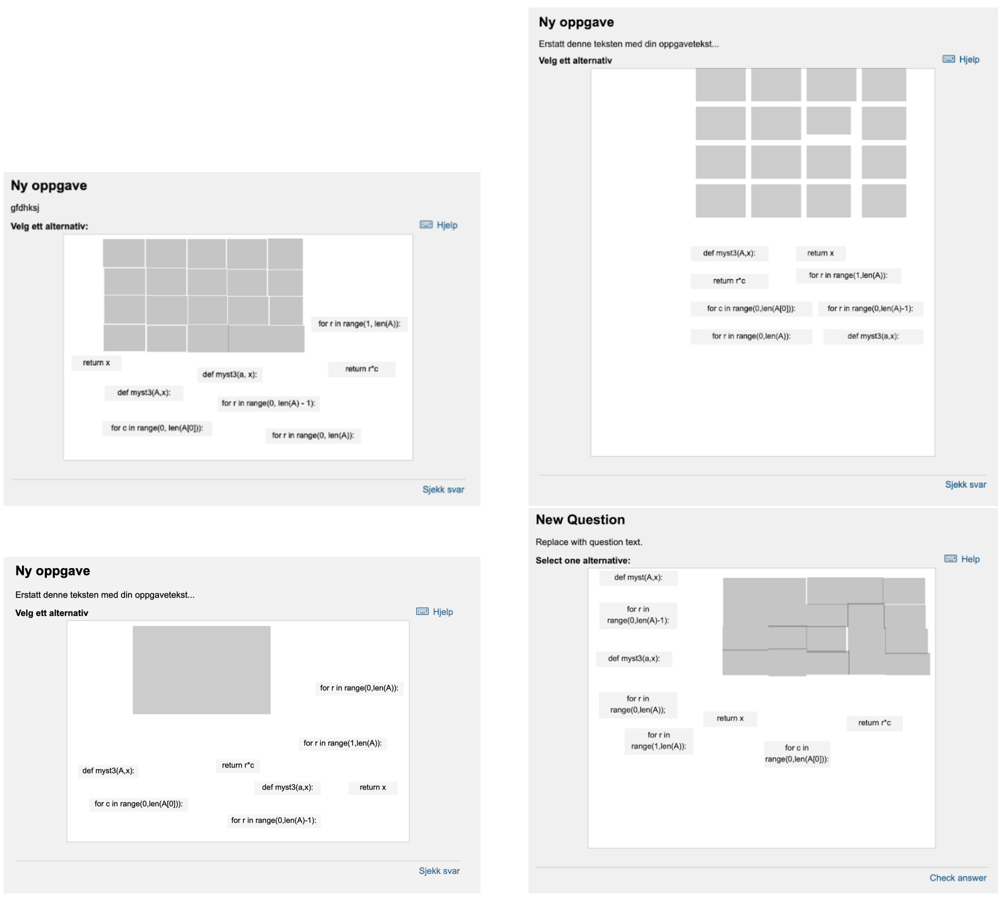

Figure 5.33: Task manually created by test participants in Inspera

and user-errors, while the final results had no design inaccuracies compared to the given tasks.

Figures 5.33 shows the outcomes of the participants manually creating Task 1 using Inspera. The resulting tasks are both inconsistent and different from the given task design they were asked to create, as shown in Figure 4.1.

SUS

The SUS questionnaire was filled out by all participants and calculated into a single score, represented in Figures 5.34 and 5.35. Using the SUS Score categories and acceptability ranges from Figure 4.4, the values established in the experiment places the perceived usability of manually creating Parsons problems in Inspera close to the adjective rating *WORST IMAGINABLE* with a mean of 28. This rating is well within the *NOT ACCEPTABLE* acceptability range. The IT artefact, on the other hand, with a mean score of 86, had an adjective rating of *EXCELLENT* and an acceptability range of *ACCEPTABLE*. The low standard deviation scores of the SUS scores also show that there was a general agreement between the participants.

SUS	Inspera	IT artefact
Participant 1	35	90
Participant 2	42.5	92.5
Participant 3	15	95
Participant 4	32.5	87.5
Participant 5	15	65
Mean	**28**	**86**
Standard Deviation	**12.42**	**12.07**

Figure 5.34: System Usability Scores gathered from test participants

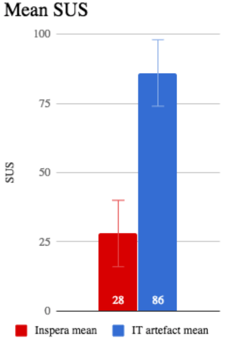

Figure 5.35: Mean SUS

Wilcoxon signed-rank test

Since it is not possible to check if the gathered data is normally distributed with this few data points, a non-parametric statistical hypothesis test had to be utilized instead. To check if the differences in time spent and SUS scores found were due to chance or not, the Wilcoxon signed-rank test (Wilcoxon, 1945) was employed. If there were no difference in the two compared data sets, the chances of getting the "+" or the "-" sign should, based on the null hypothesis stated in Section 4.3.2, be $p = 1/2$. Using this we get the following calculations.

Wilcoxon signed-rank test	App (Time spent)	Inspera (Time spent)	Sign
Participant 1 Task 1	70	620	+
Participant 1 Task 2	201	1055	+
Participant 2 Task 1	111	796	+
Participant 2 Task 2	71	444	+
Participant 3 Task 1	179	893	+
Participant 3 Task 2	175	627	+
Participant 4 Task 1	189	885	+
Participant 4 Task 2	129	550	+

Table 5.12: Wilcoxon singed-rank test on time spent

$$p(x) = \binom{N}{x} p^x (1-p)^{N-x} \tag{5.1}$$

$$p = 1/2 \tag{5.2}$$

$$p(x) = \binom{N}{x} \frac{1}{2^N} \tag{5.3}$$

Time spent:

Since time spent was gathered for each participant and for each task completed, the individual tasks were compared in the Wilcoxon signed-rank test, as seen in the Table 5.12.

With a dataset of size $N = 8$ and $x = 8$ where x is the number of positive signs, the final calculation is seen in Equation 5.4.

$$p(8) = \binom{8}{8} \frac{1}{2^8} \approx 0.004 \tag{5.4}$$

This means that there was a 0.4% likelihood that the differences found between the time spent was due to chance. Using the common statistical significance level of $p < 0.05$, the result of $p = 0.004$ can with confidence reject the null hypothesis.

SUS:

Using the SUS scores from Figure 5.34, with a dataset of size $N = 5$ and $x = 5$ where x is the number of positive changes, the final calculation is seen in Equation 5.5.

$$p(5) = \binom{5}{5}\frac{1}{2^5} \approx 0.03 \tag{5.5}$$

This also means that the differences found in the SUS scores had a 3% likelihood of being due to chance. This makes it possible to reject the null hypothesis when using the statistical significance level of $p < 0.05$.

5.2.2 Qualitative Data

The experiment phase yielded well over five hours of footage with raw data ready for qualitative data analysis. For more information about the gathered qualitative data, see Section 4.2.2. Before commencing the analysis, the footage was transcribed as a mean to ease the process of working through the data. Using the transcriptions, the data was divided into units of texts, where each unit was assigned one or more categories in an inductive approach (see Figure 5.36).

The categories were refined until a well defined list of categories emerged, which can be found in Appendix E. Then, the frequency of each category was calculated, so that the categories could be ordered by this frequency. It is important to note that these frequencies did not define the importance of each category, rather they just gave an incentive to which categories to focus on.
After the categories and frequencies were counted and added, the list was divided into two: Categories for Inspera, and categories for the IT artefact. The reason for this, was that the categories regarding Inspera was not really of interest for further development of the IT artefact.

To better distinguish categories, they were again divided into two sub groups, one for positive categories and one for negative. With this division, it was easier to distinguish between praise and constructive feedback. The positive categories were taken to heart, while the constructive feedback was used to construct a list of possible improvements of the IT artefact, which can be found in Appendix D.4. While not all of the possible improvements were doable within the scope of this thesis, e.g. 'Want to be able to generate other task types as well', the most important ones were added as tasks for the iteration following the user test. In Section 6.2.2, the qualitative results are discussed.

The three most prominent categories, in terms of frequency, for Inspera can be

TRANSKRIBERING	KATEGORI
INTERVJUER: Hva tenker du om systemet vi har laget? **TESTPERSON:** Jeg synes det var sinnsykt mye lettere å bruke enn Inspera. Det var ganske enkelt.	Ser nytteverdi av appen
TESTPERSON: Har dere tenkt på å kunne laste opp distraktorene også? Gjennom "Choose file"-knappen? **INTERVJUER:** Ja, det har vi fått tilbakemelding om tidligere. Hvordan tenker du dette burde være? **TESTPERSON:** Kan se for meg at når man skal lage oppgaven, så har man alt samlet i en fil. Spesielt hvis det blir litt mer komplekse oppgaver. Men man kan jo egentlig ikke ha så lange oppgaver med denne oppgavetypen. Eller, kanskje litt lange med denne **(peker på task2, som ikke har indenting)**, siden den bare teller rekkefølge. Men indentering er jo viktig, så det er kanskje dumt at den er slik. Men det er jo utenfor deres kontroll.	Ønsker å ha mulighet til å bruke egen editor til både kildekode og distraktorer
INTERVJUER: Hva var det beste og verste med å lage oppgaven i Inspera? **TESTPERSON:** Editor-vinduet var helt j**lig. Helt j**lig. At det ikke gikk an å få slippområdene like… Hadde de bare tatt vanlige Microsoft-funksjoner som på figurer, så kunne man copy paste for eksempel. Ser også for meg at det fort kunne blitt feil med koblingene. Vanskelig å se hvilken som går til hvilken. De **(les: Inspera)** kunne gjort dette så mye bedre. Var helt j**lig, rett og slett. Ingenting som funket.	Vanskelig å manipulere posisjon, høyde og bredde til dra- og slippområder Ikke intuitivt hvordan man kobler sammen dra og slippområde Generell irritasjon/frustrasjon på Inspera (ikke intuitivt)

Figure 5.36: Transcription snippet with assigned categories, from one of the interviews

found in Tables 5.13 and 5.14, and for the IT artefact in Tables 5.15 and 5.16.

From Table 5.14 (Negative Categories Inspera), it is clearly evident that users generally felt frustration towards Inspera when solving the given tasks. Outbursts like 'There has to be a better way of doing this?' and 'This makes no sense to me' were quite common during the user tests. Moreover, there was only one positive category that emerged from the use of Inspera, which was their preview functionality. These results clearly indicates that the process of creating Parsons problems in Inspera is not optimal.

When it comes to the categories of the IT artefact, the results where fairly fifty-fifty, with an edge towards negative categories. Most of these negative categories were in regards to features not being intuitive enough. These categories were taken

into consideration, and created tasks for in the proceeding iteration.

Positive Categories Inspera

ID	Category	Frequency
01	Bruker forhåndsvisning	1

Table 5.13: Top 1 positive categories (only one entry in the table) from Inspera, based on frequency. The complete table can be found in Appendix E.1.1

Negative Categories Inspera

ID	Category	Frequency
02	Generell irritasjon/frustrasjon på Inspera (ikke intuitivt)	21
03	Vanskelig å manipulere posisjon, høyde og bredde til dra- og slippområder	9
04	Ikke intuitivt hvordan man kobler sammen dra og slippområde	8

Table 5.14: Top 3 negative categories from Inspera, based on frequency. The complete table can be found in Appendix E.1.2

Positive Categories IT Artefact

ID	Category	Frequency
17	Ser nytteverdi av appen	7
18	Utnytter / liker hjelpefunskjon	7
19	Add task og Choose file er intuitiv	6

Table 5.15: Top 3 positive categories from the IT artefact, based on frequency. The complete table can be found in Appendix E.2.1

Negative Categories IT Artefact

ID	Category	Frequency
26	’With indenting’-toggle er utydelig og lite intuitiv	11
27	Ønsker copy paste i appen	9
28	Bruker ’Add’-knapp før input-felt	8

Table 5.16: Top 3 negative categories from the IT artefact, based on frequency. The complete table can be found in Appendix E.2.2

Chapter 6

Discussion

In this chapter, the design and creation and experiment strategy with their results will be discussed. For reference, the research questions for this thesis were:

1. How can one design and create a system for efficient generation of Parsons problems for digital programming exams in Inspera?
2. What is the effect, in regards to usability, of using the IT artefact to generate Parsons problems for digital programming exams in Inspera, compared to the manual method?

6.1 Design and Creation Discussion

The main focus of this section is to highlight what was learned during the design and creation process on how one can design and create a system for efficient generation. The content of this section is thus based on what the researchers found, and any other insights that emerged from the process. The goal of these insights is to possibly help or contribute to others looking to create systems for efficient generation of QTI-based tasks.
The applicability of these findings can, of course, be limited, due to the narrow focus of this thesis. While the main focus of the design and creation process was to try to efficiently generate Parsons problems as drag and drop tasks for Inspera, others creating similar systems for automation and mass generation might gain additional insight. Since efficiency was, as mentioned in Section 4.2.2, measured by time used, the goal was to generate these tasks with a minimum amount of waste or unnecessary effort.

Figure 6.1 shows a proposed model that provides an overview of some of the most

important aspects to consider during the different stages of creating a system for automation. The model combines some of the key elements from Section 2.6 (Automation) and Section 2.7 (Design), together with the findings of this design and creation process.

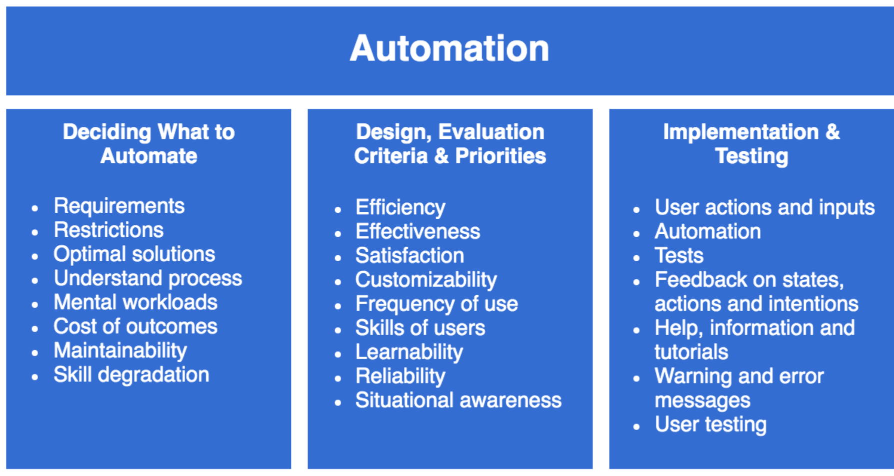

Figure 6.1: Considerations during the creation of a system for automation

6.1.1 Deciding What to Automate

These points focus on what must be examined and contemplated when deciding what to automate. These considerations can help understand which of the four-stage model of human information processing the different automation functions should be applied to (Parasuraman et al., 2000). For example, by exploring these factors, one can learn that a function originally thought of as requiring action automation, might require more decision support or analysis automation. But the main goal of these objectives is to support system developers find out if, how, and why different functions should be automated or not.

When creating a system in general, there are often specific requirements and restrictions that create a framework which one has to operate within. For this thesis, one of the main requirements and restrictions were to specifically generate and automate the process of creating Parsons problems for Inspera, using the drag and drop question type. As with most software development processes, properly gathering and specifying as many requirements and restrictions as possible, is extremely important to avoid wasting time and resources. Finding, analyzing, and documenting requirements are known as requirements engineering (RE) (Sommerville, 2011), and should be focused to get a good understanding of what the

system should provide and what restrictions must be considered.

Creators of an automation system should also focus on getting a good understanding of the function one is trying to automate, as well as finding the optimal solutions the function should output. The optimal solutions are more concerned with learning and understanding what specifies an optimal solution within the domain. For example, for this thesis, the optimal solutions would be in regards to how the drag and drop tasks had to be designed to look like a Parsons problem. In other words, the optimal solutions specifies what must be created, and by considering the ideal ways of creating this, one can better evaluate to what extend it can be automated.
For this thesis, the design (look and behavior) of the tasks to generate was predetermined, as mentioned in Section 2.4. This made it possible to clearly define the least amount of input required by a user. Generating lots of tasks efficiently means that the end-user should do as little work as possible, or reduce human operator mental workload (Parasuraman et al., 2000), and let the IT artefact do the rest. If possible, the tasks to generate should therefore be as optimal and clearly defined as possible. Spending time figuring out the best possible design for these tasks before creating a system for mass generation is therefore preferable. Another option is, of course, to improve the task design iteratively together with the development of the IT artefact. An iterative improvement might also work better with agile software development methods. The process of finding the optimal solutions were, as mentioned, out of the scope for this thesis, but the requirements regarding task design and optimal solutions are, for others creating similar systems, extremely important to efficiently generate useful tasks. So to understand how and what has to be automated, one must understand and know what output is expected from the function.

Having a clear overview of as many aspects of the task creation as possible is essential to properly understand and evaluate what aspects of the process can be automated and not. Finding the activities that causes the highest mental workload, and check if these can be automated, is one of the main criteria to improve human productivity and efficiency (Parasuraman et al., 2000). During our development, the permutation dilemma was not discovered until Iteration 4, as seen in Section 5.1.9. This was also found to be an extremely critical feature, and if it had not been implemented, the significance of the IT artefact would have been greatly reduced in the sense that many code snippets could not have been generated correctly with the IT artefact. Manually finding permutations also requires a high mental workload once the code snippets become more complicated. Thus, gathering requirements and insight from as many sources as possible can therefore help discovering the most critical features as soon as possible, and this can also be done iteratively.

The drawback of gathering requirements from multiple sources, is that the needs of each individual can be remarkably diverse. Finding the smallest set of the most critical requirements is, as found during this process, one of the greatest difficulties when creating useful and efficient mass generation of tasks. One would optimally like to implement all gathered requirements and suggested features, but in reality, this would simply increase the IT artefacts complexity and lower efficiency.

The cost of outcomes and skill degradation are both mentioned and discussed by Parasuraman et al. (Parasuraman et al., 2000). Cost of outcomes mainly looks at the risk or errors associated with different outcomes and how automation might reduce the risk, while skill degradation is concerned with the skills the human operators forget when a function is automated. Lastly, maintainability must be considered to better understand if the automated function is sustainable in the long run. For example, if the QTI-format used by Inspera was continuously updated and changed, the created IT artefact would require a lot of maintenance as well. And since the IT artefact works more like a plug-in and a tool used for a much larger system, it must continuously heed to the rules and constraints of Inspera. The maintainability must thus be considered to understand how many resources the automated function will require down the road.

6.1.2 Design, Evaluation Criteria & Priorities

These points from Figure 6.1 focus on different criteria and priorities that must be established for the design of the automation system. These considerations mainly help determine what features one would like to concentrate on and what should be established as the most important. Thus, these points should give a good checklist over the different aspects that should be given some thought when designing the system.

The first three points are the subcategories of usability, as discussed in Section 4.1. The main thing to consider here is to what extend each of these should be satisfied. Is efficiency the main aspect of the automation? Or is the goal the reduction of human made errors? While efficiency, effectiveness, and satisfaction might overlap in the sense that a more efficient system, for example, might cause more satisfaction among the users, the overall correlation is found to be weak (Frøkjær et al., 2000). All these points are here to help illustrate what the system needs, in terms of the most important features. For example, extra focus on efficiency might require a reduction of required human actions or inputs in the system, an optimized algorithm, or an investigation of which manual functions are the real bottlenecks. Effectiveness might require the developers to focus more on learning where most errors occur and how they can be removed by automation. Lastly, improving sat-

isfaction might involve more focus on how to create a user-friendly and intuitive system.

Another important reason to clearly define to what degree these should be satisfied, is that they might have distinct effects on other aspects. During this research, it was found that improving efficiency comes at the expense of customizability. The final IT artefact only supported two different design options (2D and 1D Parsons problems), but during the development process, multiple other needs and design suggestions surfaced, as seen in Appendix D.4. Many of these suggestions would clearly require much more functionality and options in the IT artefact, which would influence the time spent. For example, the addition of permutation support almost doubled the amount of input a user had to make. This led to the quite obvious conclusion that when automating a process, efficiency comes at the expense of customizability.

As stated in Research Question 1, the goal of the thesis was to create a system for *efficient* generation of Parsons Problems. For this reason, it was decided to prioritize efficiency over customizability. Less required user input, less options and less customizability helps improve the ease of use, learnability, and intuitiveness of the IT artefact since there is less things to learn and more information given about the remaining features. During user testing, one of the participants said the following: *"I greatly prefer systems that performs one specific task incredibly well, instead of a system that tries and fails to do multiple things at once"*. The researchers believe this quote clearly reflects how one needs to approach the creation of an efficient system. Efficiently generating tasks also means that the end-user should not spend unnecessary time with the IT artefact.
Since an end-user is assumed to only use the IT artefact a handful of times during a year, it is extra important that it is intuitive and easy to use. Here comes the concept of considering the frequency of use in a system. In more complex and larger systems, one can expect the user to spend additional time learning how to operate it, but in this situation, the IT artefact should optimally work as an efficient tool that can be used when needed. And in this case, the system must be designed so one can easily understand how to operate it again after a year without use.

Considering the skills of the users also help define how much explanation is needed to the different actions being performed by the system. For example, for this IT artefact, the end-users were highly competent course-supervisors that have decent knowledge of Inspera and programming exams. One can in this case, for example, assume that a directed acyclic graph (DAG) does not need to be explained to the users. Clearly defining and recognizing the target audience might help improve all aspects of the system.

Parasuraman et al. also discusses how reliability, complacency, and situational awareness must be considered when automating a function (Parasuraman et al., 2000).

6.1.3 Implementation & Testing

The final considerations in Figure 6.1 gives an overview of a few tools that can be used to adjust or include some of the previously discussed points. For example, to increase reliability, one can include more system, acceptance, integration, or unit tests. And to improve efficiency, improving or reducing the number of user actions and inputs might help. Proper feedback on states, actions, and intentions also help improve the situational awareness of the user and might reduce skill degradation (Parasuraman and Riley, 1997). All these considerations were used and discovered during this thesis, and while they are in no way revolutionary, they might provide additional understanding or an overview of how things can be implemented.

Since the expected frequency of use for this IT artefact was low, learnability and ease of use had to be focused. To achieve this, a description or help function was implemented for multiple features to eliminate any source of confusion as of what a feature did. In addition, providing detailed and immediate feedback when performing actions in the IT artefact greatly improved users understanding. In general, the user testing showed that the IT artefact lacked some initial feedback to properly show the test participants what was going on and what different actions did. Some had problems understanding what was being generated and how it was connected to Inspera. As mentioned, providing situational awareness (Parasuraman et al., 2000) and proper feedback about the automation's states, actions, and intentions (Parasuraman and Riley, 1997) is crucial. The feedback should give good indications as to what will be generated by the IT artefact, making it possible for users to test out different actions and see what happens. This can greatly influence the perceived ease of use, as well as reducing errors when one can clearly see what will be generated. For example, during the initial user testing, the lack of feedback from the 2D Parsons checkbox caused multiple participants to forget this aspect of the task. With the implementation of a task preview, one could now discover such overlooked features much easier and earlier than before.

Proper warning messages or alarms is also important when a dangerous situation or errors occur (Parasuraman and Riley, 1997), and this was implemented to a certain degree. For example by providing error messages if the task text description had too much formatted text, if too many permutations could be generated, or if something went wrong during the task generation. There could still be added more warning and error messages to properly cover everything that could go wrong, but

the current level of messages was satisfactory for a prototype.

6.1.4 Other comments

A drawback of having to create an automated process is the need for domain knowledge, at least during the initial requirement gathering. Great effort has to be put in to automating a process, and having domain knowledge can greatly reduce the work and time spent developing such an artefact. Designing optimal Parsons problems for Inspera requires deep insight and knowledge of what is supported by Inspera, and what is optimal and intuitive for students. The generation of QTI also requires knowledge of its rules and features, and what Inspera supports here as well. The advantage of the automation, is that this domain knowledge is needed in a much less degree after the initial development of the generation tool. By having experts within the field determine how things should be generated, future users do not have to spend extra time trying to learn and find optimal ways of creating these tasks. But as mentioned, this might also cause skill degradation (Parasuraman et al., 2000), since course supervisors no longer has to think of how to design their exam questions. Skill degradation could also come in the form of not knowing how to manually create tasks in Inspera, at least if the IT artefact is further developed into a more comprehensive system for generation of tasks for Inspera. But as of now, the researchers do not consider it a big problem if a course supervisor forgets how to manually create drag and drop tasks, since this can be easily learned again with Inspera.

There are some shortcomings and flaws connected with the created IT artefact that can potentially reduce the usefulness and value it brings to the end-users. While the IT artefact supports the creation of both 2D and 1D Parsons problem, the permutations were only implemented for the latter task type. The consequences of this is that users might feel restricted to only create 1D Parsons problems, because of the value that comes with the implementation of permutations. The IT artefact still provides a preview of permutations when a 2D Parsons problem is selected, but these permutations will not be implemented in the generated task. This makes it possible to still get a good indication of the soundness of the task. But not having proper support for permutations in a 2D Parsons problem can decrease the IT artefacts value. Another problem is that a user is able to copy paste highly formatted text to the task description, which can have unknown consequences for Inspera. Other, undiscovered bugs might also have severe consequences if the users develop too much complacency and does not check the generated tasks. So while efficiency was the main focus when creating the IT artefact, providing good test coverage, properly implemented features, and as few bugs as possible is still essential to the overall usefulness of the system. One can not simply focus on

efficiency, since there are many other non-functional requirements that must be considered while developing an IT artefact.

Initially, there was thought to be few features in the application, and as a result pure JavaScript was used in development. Later, as the number of features expanded substantially, it is obvious that having a framework would have been more beneficial. One thing is having cleaner/better structured code, but another benefit would be the ease of testing. Frameworks usually provides testing frameworks made for them, making the testing process much easier.
In addition, frameworks are usually easier to scale than pure JavaScript code. With the continuous growth of features, the source code got a bit more unstructured for each feature added. With a framework, such as React, the addition of features would mean the addition of more components, which accommodate scaling. Writing pure JavaScript, new features meant that the entire source code had to be revisited to be sure all relations were still intact. In our case, since the IT artefact was developed as a prototype, the scalability aspect was not a part of the initial requirements. But for further development of this IT artefact, or for others planning to create similar systems, it is necessary to spend some time to properly plan out the actual scale of the system and how that influences technology choices and code architecture.

Another concern regarding maintainability was the QTI Converters dependency on the skeleton obtained from Inspera. If Inspera at a point in the future decides to alter the structure of the QTI version, add features, or remove features, the IT artefact will be rendered useless, or non-compatible with Inspera, if it is not maintained and updated to reflect said changes. During the development, the researchers also encountered some of the IMS QTI shortcomings mentioned by Piotrowski (Piotrowski, 2011). This was mainly at the beginning of the development, when the developers first tried to learn and understand which of the IMS QTI features were supported by Inspera. Although Inspera supports IMS QTI v2.1, it does not support all available features, making it difficult to clearly find the ones that are truly supported. This was mainly why the developers chose to reverse engineer the XML files Inspera generated and use those as skeleton code instead. The lack of proper interoperability using IMS QTI, at least with Inspera, was also evident when the developers tried to create drag and drop questions using TAO from Section 3 (Related Work) and upload this question to Inspera. The uploaded question could not be opened in Inspera at all, for unknown reasons, even though they used the exact same IMS QTI versions. So based on the design and creation of this IT artefact, the developers recognize and agree with many of the shortcomings mentioned by Piotrowski.

Overall, the crated IT artefact shows that the domain of generating multiple QTI-

based tasks for digital programming exams, can be highly automated. Not only does the IT artefact efficiently generate multiple tasks, but the user-testing shows that it generates single tasks more efficiently as well. The automation also reduced the possibility of human errors when creating the tasks. Highly repetitive work, that is, creating tens or hundreds of tasks, have a higher probability of human error. Such accidents can have severe consequences, which automation can reduce the possibility of. But as mentioned in Section 2.6, a seemingly reliable automation system can quickly lead to too much complacency (Rovira et al., 2007). While the IT artefact might reduce the amount of human errors made, it might not be completely free from bugs or errors, which can cause undiscovered failures. To properly address this issue, the IT artefact should be tested to a much greater degree by more people and supplemented with more integration, unit, system, and acceptance tests. This will be further discussed in Section 7.2 (Future Work).

6.2 Experiment Discussion

In this section, quantitative and qualitative data will be discussed and interpreted in light of the research questions for this thesis. It is important to note that one of the largest drawback of the data gathered from the experiment, is that the data not necessarily reflects the IT artefact in its latest version. The experiment was conducted at the end of Iteration 4, as discussed in Section 5.1, and several features were changed or added after this iteration. The IT artefact withstood extensive changes during Iteration 5, which means that the data discussed in this section only reflects the initial user testing conducted in Iteration 4.

6.2.1 Quantitative Discussion

In this section, the quantitative data results from the experiment will be discussed and interpreted with regards to Research Question 2. Before these data points are discussed, some of the main flaws must be acknowledged. The created user testing was, as mentioned in Section 4.1, conducted in a simulated and context-restricted environment. The tasks to complete were also created by the researchers, who knew the IT artefact would generate these tasks correctly. Comparing the inaccuracies in the different solutions, for example, can therefore be seen as a measurement highly stacked against Inspera. The IT artefact was prepared to handle these test cases, knowing the tasks would only require what it was able to create. In real-world situations, tasks with different features and details might be desirable, meaning the IT artefact might be incapable of creating them. The image we get from the gathered data points, does therefore not necessarily reflect the real-world use of the IT artefact. But the created tasks were developed together with the

thesis supervisor, and assumed to be authentic and reflect expected use as closely as possible. So while the tasks might properly reflect most use cases, some course supervisors still might have different expectations regarding the IT artefact.

In general, it is important to make sure that the desired features and details regarding a task to be generated is clearly defined, at least when one is trying to automate the generation of said tasks. In this research, the final look of the tasks to generate was clearly defined, as shown in Section 2.4. The IT artefact could then focus on generating the tasks based on these requirements instead of having to accommodate for further customizability. Mainly, the quantitative data gathered only reflect the reality as long as the generated tasks are relevant and exactly as an end-user wants them. During the user testing, these needs and wants were in a sense forced on the participants when they were given the pre-defined tasks. Thus, the gathered data assumes that this pre-defined design of a task reflect future end-users requirements as well.

The quantitative data gathered was also based on few test participants, which questions the validity of all generalizations drawn from this data. One can from this data primarily see indications of what effects the use of the IT artefact will have when generating Parsons problems. To draw more legitimate conclusions, more user testing has to be performed.

The data can not be generalized to cover other tasks in Inspera either. The creation of Parsons problems in Inspera using a drag and drop task is quite distinctive, meaning that other task types in Inspera not necessarily share the same usability effects of being automatically generated. Since other task types than drag and drop is out of the scope of this thesis, no further analysis will be made regarding other tasks. But it is still important to note that these findings does not reflect all tasks in Inspera, and that they would require their own examination.

Time spent

As seen in Figure 5.30, time spent solving the tasks using the IT artefact was at approximately one fifth of the time spent using Inspera. More accuratly, the mean time spent in Inspera and in the IT artefact had the ratio of 5.22, which were extremely close to the KLM calculations (4.82) as well. The KLM calculations show that by objectively looking at the process (by simply counting the number of keystrokes and buttons that has to be pressed), one can see that the created IT artefact will be more efficient than manually creating the same task in Inspera. The same results were found during the actual user testing, even though the test participants spent a substantial amount of time figuring out what to do next and how

to achieve the wanted results. All of the participant had some previous experience with Inspera, but no one had experience creating a drag and drop task. The previous experience was though to be a small advantage for Inspera, but based on the results, any small advantage that might have existed does not seem to be of any importance regarding the time spent.

As one can also see from the Wilcoxon signed-rank test, the difference in time spent rejected the null hypothesis with a $p = 0.004$ and a significance level of $p < 0.05$. While this signed-rank test does not consider to what extent the data differs, it only checks to see if the data differs, and in what direction. But these measurements mean that one can with high likelihood say that there will be an improvement in efficiency when using the IT artefact instead of Inspera. And in relation to Research Question 2, this gives a good indication that one of the effects of using this IT artefact, is an improvement in time spent when creating the same tasks. It is still important to remember that the validity of these p-values can be questioned due to the low number of participants. The difference in time spent will also be highly dependent on the size of the tasks given. For example, a task with just three lines of code would probably generate less time-difference than a task with 10. And if a task was small enough, one might encounter less time spent using Inspera. So the gathered data might be a result of the size of the tasks given and the low number of participants. But during this research, both these aspects were considered to be representative of the real-world tasks and users of the IT artefact, and to properly eliminate or resolve these concerns, further testing should be conducted.

The differences found in time spent are clearly evident, but still considered to reflect the effect of the IT artefact in a considerably small scale. As mentioned in the introduction of this section, the data gathered during the experiment only reflected the state of the IT artefact during Iteration 4, as seen in Section 5.1. In Iteration 5, the IT artefact was capable of generating multiple unique tasks given the same algorithm, using different subsets of distractors. This additional feature of generating multiple unique tasks at the same time would greatly change the time differences between the two processes. The user testing simply compared the processes in a 1-to-1 manner, by creating two task in Inspera and two tasks in the IT artefact, but the additional feature of generating multiple tasks would greatly skew the ratio. Using the IT artefact, only one task has to be created to generate hundreds of unique tasks (given enough distractors), while in Inspera all tasks must be generated manually. It is possible to create dupliactes of a task in Inspera as well, but the user still has to open each task and manually change the correct distractors to another unused subset of all distractors. This means that the results found during the experiment only shows the least amount of time the IT

artefact will save an end-user, while any further generation of tasks will increase time saved.

During Iteration 5, the permutation support was added as well, which could further influence time spent in the IT artefact. Although this was not tested and validated, the assumption was that the time it takes to create a DAG to make the IT artefact generate all possible permutations takes much less time than trying to find and implement all these permutations manually. In general, the new features implemented in Iteration 5 are all assumed to improve the efficiency and effectiveness of the IT artefact, but it is important to mention that this is not verified and should be tested in the future (See Future Work 7.2).

Research Question 2 looks at the effect of the IT artefact, in regards to usability. But as stated in Section 4.2.2, the time spent (efficiency) can not represent usability by itself. The effectiveness and satisfaction must also be taken into consideration before one can get a clear understanding of the IT artefacts usability compared to Inspera. But the efficiency aspect of the usability can with certainty be seen as a obvious and evident improvement. With the Wilcoxon signed-rank test, one can with confidence see that the IT artefact will decrease the time required to create a task. And although there were quite few data points, the discovered time spent ratio of 5.22 between the IT artefact and Inspera also give an indication to the improvements extent. In a broader sense, this shows that the generation of multiple tasks has the potential to be highly automated and and time-saving as long as the generated task design correctly reflect an end-users requirements.

User-Errors, Inaccuracies, Questions and System-Errors

In this section, the data points from Figure 5.32 from Section 5.2.1 are examined. A flaw with some of these data points, is that the user-errors and questions might simply reflect initial confusion regarding Inspera and the IT artefact, and not necessarily be accurate once a test participants learns the basics. In that case, these data points simply reflect the initial ease of use and how intuitive the two systems are.

Section 4.3 gives a definitions of how these data points were defined and gathered. All of these data points, except the System-Errors, had clear improvements when using the IT artefact. Research Question 2 focuses on the effects of the IT artefact regarding usability, and these data points reflect the effectiveness aspect of the usability. The inaccuracies show the quality of the solution itself, while the other measurements are based on the process of creating the tasks.

As mentioned in Section 4.3, the counted inaccuracies was defined as one or more

of the following categories:

- Inaccurate drop-area positioning and alignment
- Inaccurate drop-area sizes
- Inaccurate drop-area sizes
- Inaccurate drag-area positioning and inaccurate drag-area sizes

These inaccuracies are better shown in the Figure **??**, **??**, 5.33, and **??**, which are the screenshots of every Task 1 that was manually created during the user-tests. The correct look of Task 1 can be seen Figure 4.1. This clearly shows that none of the created solutions actually looked like the given task, so even though it took the participants more than 4 times longer to complete the tasks, the results were still sub-par and not fit for use in an exam. The test participants were aware of the inaccuracies, but were not willing to improve the results further. This was usually due to frustration and fatigue. The frustration with Inspera was evident during the testing, and is described in more detail in Section 6.2.2. But all the data points from Figure 5.32 can give a good indication as of where some of the frustration and fatigue originated.

The data show the initial and potential errors and confusion an automation of this domain can remove. Automating a domain is generally known to reduce errors and make results more predictable, and the gathered data points here support this fact. The most time-consuming and 'brainless' work of creating a Parsons problem in Inspera is the re-positioning and re-sizing of drag and drop fields, which are all automated using the IT artefact.

The system-error count was slightly higher with the IT artefact, which mainly came from bugs not discovered by the developers. For example, even though it worked in the development build, the release build of the IT artefact did not support copy paste functionality. This was because the developers had forgotten to explicitly define it. The bugs and system-errors encountered were all fixed after the testing. There still might exist undiscovered bugs in the IT artefact, which potentially could affect several generated Parsons problems, but given the unit tests and passed use cases, the number of bugs was hopefully kept to a minimum. Since a small bug can potentially affect many generated tasks and potential exams, providing good test coverage of this automation is extremely important. During the development of this prototype, testing was focused on, but if the goal would be to create an end-product, testing should play a much larger part of the development. Creating an end-product and assuring an error-free IT artefact was not the goal of this thesis,

but can still be seen as one of the main limitations of the current IT artefact. As the research questions state, the main goals was to see how a system for efficient generation would be created, and the effect of this system.

Given these data points, it is clear that the use of the IT artefact improves the effectiveness aspect of usability, by lowering the number of errors and inaccuracies made by the user, and questions asked. Even if these measurements might only reflect the behavior of novice users in both systems, it still gives good indications regarding differences in intuitiveness, ease of use, and learnability. Based on this user testing, the indicated effect of using the IT artefact, with respect to the effectiveness aspect of usability, was the following:

- Consistency in results was improved, with less inaccuracies made to the actual look and feel of the created task.
- Less confusion and less question asked when a system specializes in the automation of the specific tasks.
- Less human errors made and less domain knowledge needed since IT artefact presumably generates 'optimal' task design (which was pre-defined for this thesis) based on other expert decision makers.
- Bugs might influence more tasks and is thus more critical to find and eliminate.

SUS

The final aspect of usability focuses on satisfaction, which during the experiment was measured using the System usability scale. This section references Figure 5.34 from Section 5.2.1. These results were fairly consistent among all test participants and both systems. They also greatly reflect the observed frustration with Inspera during the testing. With this data, there was no doubt that having a specific IT artefact generating Parsons problems was highly preferred compared to having to create the same tasks manually. The Wilcoxon signed-rank test also resulted in a $p = 0.03$, which rejected the null hypothesis when using a statistical significance level of $p < 0.05$. This makes it possible to say there will with a very high likelyhood be an improvement in perceived usability when using the IT artefact instead of Inspera.

The main threat to validity regarding the SUS scores is possibly that all the test participants tried both Inspera and the IT artefact before answering the questionnaire. This means that the SUS scores might reflect the IT artefact in comparison to Inspera, instead of reflecting the IT artefact independently. The IT artefact might

have gotten much worse SUS scores if the test participants had not tried the even 'worse' alternative. For future testing of the IT artefact, one should thus consider using two separate test groups for testing the IT artefact and Inspera.

The actual design of the tasks to generate was pre-determined, as previously mentioned in Section 2.4. It is of course, possible that the confusion and frustration evident from the test participants was not only due to Insperas' user interface, but due to the pre-determined task design. The time-consuming nature of creating these tasks might be due to misuse of Insperas' drag and drop task type. If the drag and drop tasks type was created for completely different use, it can of course be problematic to force them to fit and work as Parsons problems. The main problem might therefore come from the attempt to adapt an unfit task type to a Parsons problem. It is also important to remember that these ratings does not mean that Inspera in its entirety deserves such a SUS score, but simply the process of creating these pre-defined Parsons problems in Inspera is extremely frustrating for the users.

The frustration evident in the SUS data does still give good evidence that the created IT artefact has clear effects on perceived usability. The automation of this specific domain greatly eliminates frustration, annoyance, errors, inaccuracies, and time-consuming work that must be performed in Insperas' user interface to obtain the same results.

6.2.2 Qualitative Discussion

In this section, the qualitative data gathered from the experiment will be discussed. The goal of the qualitative data was twofold, (1) to find and gather more requirements and find possible improvements, and (2) to get a better understanding of the perceived efficiency and usability of the IT artefact. Since the main results gathered after processing the raw data was a list of categories together with a description, the most important categories will be discussed here, together with the overall themes that emerged.

Delving into the categories, the one that stood out the most (twice the frequency of the next on the list) was the overall dissatisfaction and frustration with Inspera. Over the course of all the experiments, a trend emerged where all of the participants, with no exception, demonstrated some sort of irritation or confusion towards Inspera. This, alone, made it evident that the need and desire for either an improvement of Inspera, or an automated process, existed. Based on this frustration, it was clear that creating Parsons problems using the drag and drop question type manually, was extremely undesirable.

There were several functions that were not intuitive, e.g. how to connect drag and drop areas, adjust the size of the task window, and fill in text into the drag areas. All of these struggles accumulated into an overall dissatisfaction and irritation with Inspera. Some users had to spend time playing around with the functionalities, feeling they wasted time. Initially, the 'generally annoyed/frustrated with Inspera' category was split up into smaller categories such as 'finding Inspera cumbersome', 'expressing frustration', and 'feeling waste of time', but Inspera itself was not the focus of this experiment, and therefore the specific annoyances was not of great importance.

Using the IT artefact, on the other hand, the participants felt a sense of efficiency. All of the tasks were completed without any irritating events. Even though there were some bugs in the IT artefact, the stress level did not come near the process of using Inspera. The test participants still provided a great deal of feedback regarding the IT artefact and did not hesitate to comment if something was confusing. The feedback gathered during the experiment were extremely helpful and led to a long and detailed list of further improvements.

Once participants had tried both processes, they immediately appreciated the value of the IT artefact. Not only did the time spent favour the IT artefact, but the ease of use as well. After trying both methods, all participants expressed satisfaction of using the IT artefact over Inspera. If they were to create Parsons problems for an exam, they all said they would prefer to use the IT artefact. It is important to remember that the positive feedback received could also come from the Hawthorne effect (Adair, 1984) and how the test participants wants to be friendly to the researchers and their project.

Overall, most of the data gathered from the qualitative data generation are represented as requirements and discovered improvements in Appendix D. And most of the categories discovered for the IT artefact was concerned with specific improvement that could be made, and needs no further discussion here.

During the experiment, all test participant tried both the IT artefact and Inspera, and as mentioned, this could possibly have negative effects on the results. The IT artefact might have gotten more positive feedback and higher perceived usability than it deserved, since it constantly was compared to a much worse process. Another option, that could have avoided this problem, was to test the two processes on two completely different test groups. But since this would require more test participants, it was not done in this experiment.
Another problem with the perceived value, is that it was only based on the specific tasks they were given during the testing process. The perceived value and usability of the IT artefact could have gotten different results if the participants were to

come up with their own exam questions to be generated. The fundamental requirements for the Parsons problems were in this case formulated by the researchers, and not by the actual test participants. Once the end-users spend more time thinking about what tasks they want to create, a lot of new and different requirements might come up. For example, maybe the need for partially solved task proves to be much higher than assumed during this research. So mainly, it is possible that a large amount of requirements were undiscovered or ignored since the test participants were given specific tasks to create. The data gathered is thus only based on these tasks, and since the test participants was not given a chance to come up with their own exam questions, this data does not necessarily reflect the actual needs of the end-users.

As previously mentioned, Inspera with its drag and drop tasks were not made specifically for Parsons problems. Inspera is a big environment, and supporting every kind of task would most likely require too many resources. The IT artefact is designed specifically for creating Parsons problems using the drag and drop question type, and then it would only be natural that such a specific application would outperform Inspera. The experiment also simply compares Inspera with the IT artefact, while there might exist other environments where one could create drag and drop tasks and export them as QTI. Since none of the systems found in Section 3 (Related Work) generated drag and drop tasks better suited for Parsons problems or more efficiently than Inspera, this aspect was ignored during the experiment.

During the experiment, some participants also expressed a concern regarding the usefulness of Parsons problems as drag and drop tasks over other, simpler task types. Section 2.1 discusses and show that Parsons problems are just as effective, but more efficient than regular code writing tasks (Ericson et al., 2017). And the main concern from the test participants were not the Parsons problems themselves, but the use of the drag and drop question type in Inspera. The test participants found the question type unfit due to some of the inconveniences already mentioned in Section 2.1, such as the unintuitive anchor of a drag area, no feedback when hovering a drop area with a drag area, and no good support for longer lines of code. Since using the drag and drop question type was a prerequisite for this research, this concern will be discussed further in Section 7.2 (Future Work).

Chapter 7

Conclusion and Future Work

7.1 Conclusion

With the introduction of Inspera and digital programming exams at NTNU, the use of Parsons problems have become more appealing, with benefits such as automatic assessment. These Parsons problems can be created using the drag and drop question type, and make it possible to utilize automatic assessment. This thesis proposes an IT artefact that tries to automate the generation of these Parsons problems to improve usability and make the process of creating the tasks more efficient for the course supervisors. This thesis also looks at the effects the proposed IT artefact has on usability compared to manually creating Parsons problems using Inspera.

Research Question 1: *How can one design and create a system for efficient generation of Parsons problems for digital programming exams in Inspera?*

Together with the proposed IT artefact, the researchers have developed a model (Figure 6.1) summarizing the main considerations and factors found for designing and creating an automation system to, for example, improve efficiency. These factors were discovered during this research and placed in a model to provide an overview of the aspects the researchers recommend other developers consider when trying to create similar systems. The developed IT artefact also proposes a solution to efficiently generate Parsons problems for Inspera. Through multiple iterations with development, user testing, and feedback, a functional prototype was created by the researchers, and to assess its efficiency, it was evaluated with an experiment strategy.

Research Question 2: *What is the effect, in regards to usability, of using the IT artefact to generate Parsons problems for digital programming exams in Inspera, compared to the manual method?*

To evaluate the effect of the IT artefact in regards to usability, the researchers compared the process of creating specific Parsons problems using the IT artefact and using Inspera with five test participants within the sampling frame. The data generation methods used was observations, interviews, and questionnaires, which made it possible to perform both quantitative and qualitative data analysis. Although the low number of participants had its implications in regards to the validity of the data, the IT artefact showed evident improvements in both efficiency, effectiveness, and perceived satisfaction among the test participants. In regards to efficiency, the time spent using Inspera was, for every single participant, over 4 times more than with the IT artefact. The number of errors made and inaccuracies in the created solutions were also drastically reduced using the IT artefact. Lastly, Inspera received a mean satisfaction score of 28/100, while the IT artefact received a score of 86/100. So when comparing these two processes for creating the same Parsons problems, the IT artefact resulted in an improvement in all mentioned aspects of usability.
While the qualitative data gathered provided valuable feedback and ideas to further improve the IT artefact, it also captioned the frustration emerging when participants used Inspera, which further emphasized the need for the IT artefact. Overall, the test participants expressed a positive attitude towards the proposed IT artefact while the qualitative data also indicated positive changes in usability.

7.2 Future Work

During the course of this thesis, the researchers learned a great deal about the domains of digital exam creation, Parsons problems, and application development. Unfortunately, because of time limitation, not all aspects could be pursued fully and are therefore left for future work. Below are possible areas of future work described.

Generation of other tasks

The IT artefact could be expanded to generate more than just Parsons problems. One could potentially add the generation of other task types for programming exams, such as multiple choice or matching/pairing questions for Inspera. Additional task types should be investigated and evaluated properly to assess if further automation of their creation process would be beneficial or not.

Improve Permutations

As mentioned in Section 5.1.7, the chosen design of the user interface for letting users create a DAG was based on the little amount of time left of the project. Given more time, a more user friendly and intuitive interface might have emerged.

Currently, also mentioned in Section 5.1.7, the IT artefact does not support permutations for 2D Parsons problems. This is certainly a feature that would improve the usefulness of the IT artefact, and worth looking into.

Variations of the Tasks Produced by the IT Artefact

There are a few variations of the drag and drop Parsons problems produced by the IT artefact that is worth looking into in future work. For example partial code completion tasks, where parts of the problem solution is given, or multiple code lines per drag area, where two or more code lines are merged into one drag area. These variations can have several benefits, like removing "free" points and/or reducing the number of permutations of code lines. "Free" points refers to code lines that are obvious where to place, e.g. function definitions or return statements. And having code lines locked into a position, or having two code lines merged together, reduces the number of drag areas to be placed by the user, which in turn reduces the number of permutations possible for the task.

Parsons problems With Other Task Types

As mentioned in Section 6.2.2, some users expressed a concern regarding the usefulness of Parsons problems as a drag and drop task, over other task types like for example dropdown menus or copy and paste a given set of code lines into a code editor. The drag and drop variation was a prerequisite for this thesis, but the researchers believe that other variations are also worth looking into. For instance, a new question type named 'Inline Gap Match' was released in Inspera during the development of the IT artefact, which might prove useful in the future.

Validation and Feedback

Future work on the IT artefact should consider improving validation done and feedback given by the IT artefact. As this was only a prototype, not all aspects of these areas were fully implemented.
As mention in Section 5.1.6, the IT artefact allows for formatted text to be pas-

ted into the task description of each task. A future version of the prototype should either remove the unwanted formatting, or display an error message whenever such formatting is present.
Additionally, feedback in the form of error messages should be more user friendly in future versions of the prototype. Currently, some of these messages are printed in the console log in the developer window of the application, but this should preferably be moved to the application itself so that users are aware of any errors, which can provide more insight and situational awareness.

More Testing

As mentioned in Section 6.1.4, the IT artefact should be tested to a much greater degree by more people and supplemented with more integration, unit, system, and acceptance tests to avoid having complacency lead to undiscovered errors (Rovira et al., 2007). A possible solution to this would be to migrate the code to a framework, which might support a better testing environment.

Additionally, another round of user testing should be performed in the future, as features introduced after iteration 4 have not gone through proper user testing.

Other Improvements

In the list of requirements (Table D.1) and list of improvements (Table D.4) in Appendix D, one can find additional features that was not implemented during this research, but that can be made to further improve the IT artefact. The researchers suggest using these neglected features and requirements as the basis for any future work on the IT artefact, since these are already based on feedback from end-users. Alternatively, one could also choose to gather new and updated requirements. The most prominent features among the remaining ones, might be the addition of persistent storage or a tutorial.

Appendix A

Models

A.1 Activity Diagram

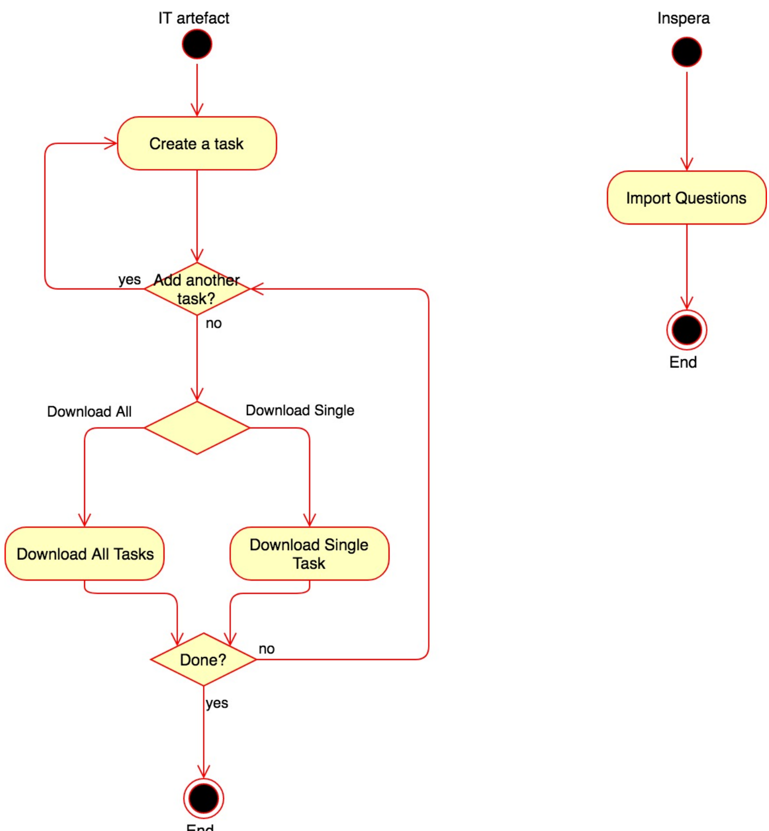

Figure A.1: The activity diagram for the process of creating a task

A.2 Development View

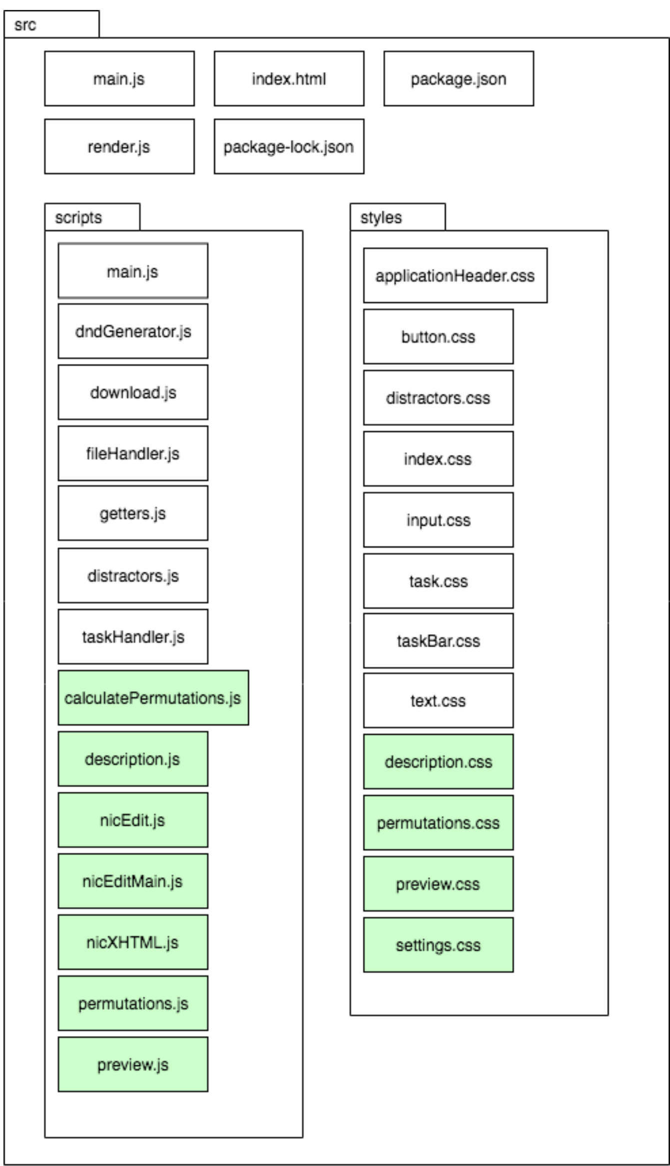

Figure A.2: Development view of the source code. The highlighted boxes are files added after the user test

A.3 Logical View

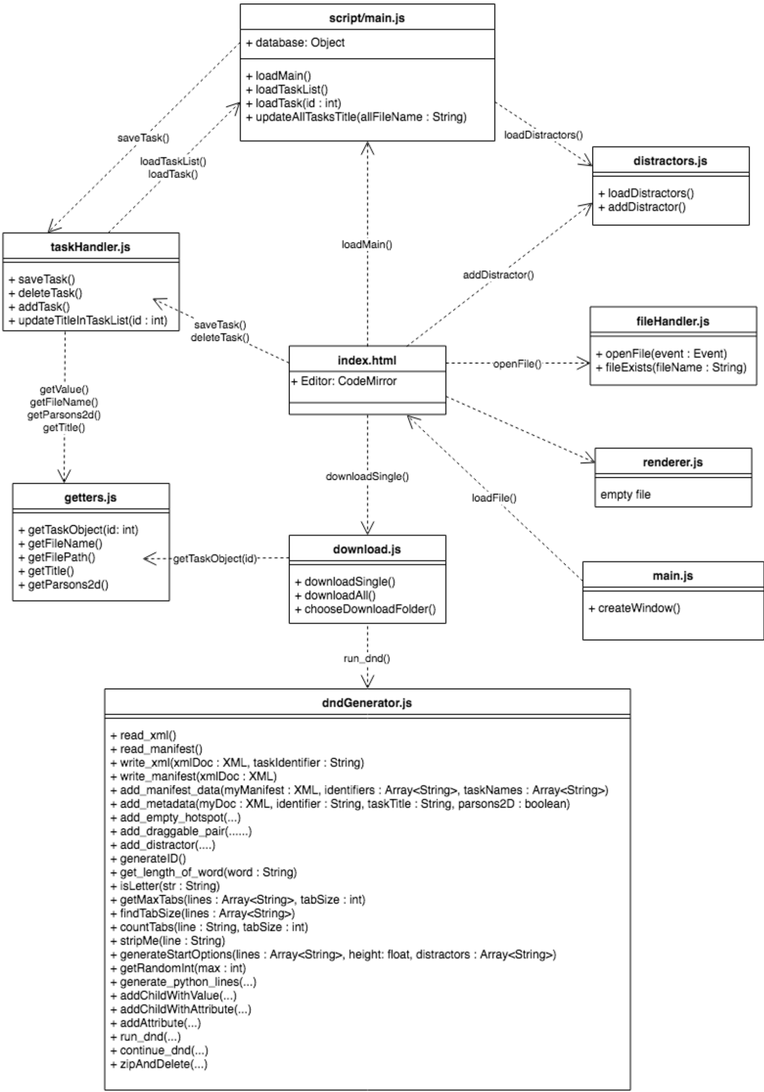

Figure A.3: The logical view of the source code before the user test (iteration 4)

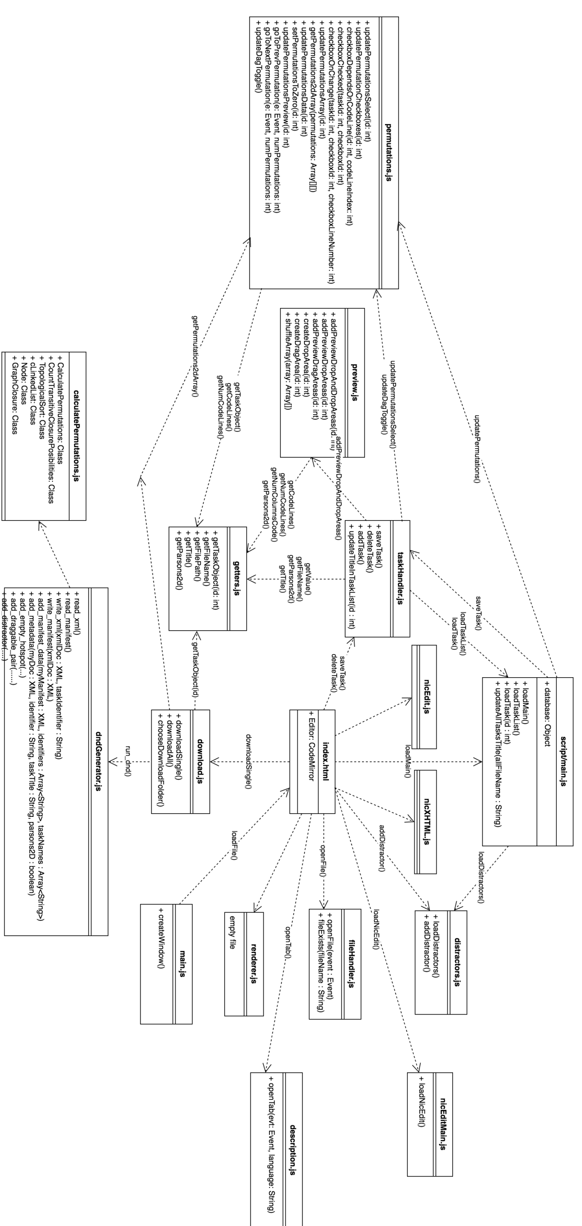

Figure A.4: The logical view of the source code after the user test (iteration 5)

Appendix B

User Manual

This user manual describes how to use the Parsons Generator application and how to upload tasks to Inspera.

B.1 Parsons Generator

B.1.1 Add New Task

On application launch, an empty window without any tasks will be loaded. There are only two actions available: 'Add task' and 'Export All'. 'Export All' will be explained in Section B.1.5.

The button 'Add Task' (refer to 1 in Figure B.1) instantiates a new task, and the newly created task will be displayed in the application. If a task is already displayed, 'Add Task' will display the newly created task.

B.1.2 Navigate Tasks

On the left hand side of the application a list of all created tasks are displayed (refer to 1 in Figure B.2). If no tasks are created, the list will be empty. If there are one or more tasks, the currently selected task is highlighted and a light blue decorator appended.

To select and view another task, click on a task in the task list.

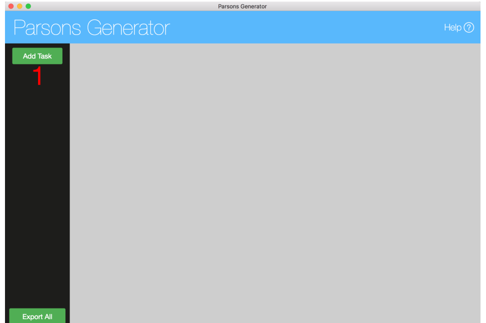

Figure B.1: Application Launch on launch

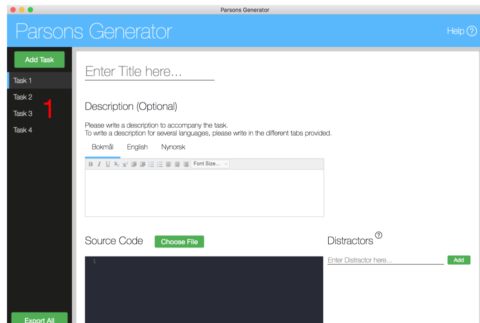

Figure B.2: Application with a list of tasks

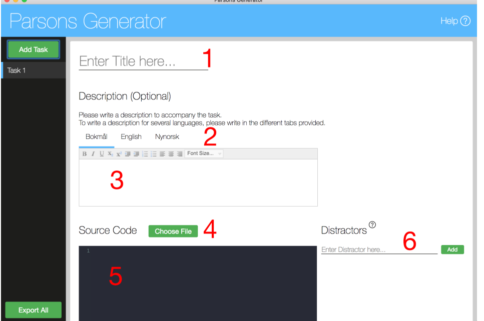

Figure B.3: Application with an empty task (Top of the task page)

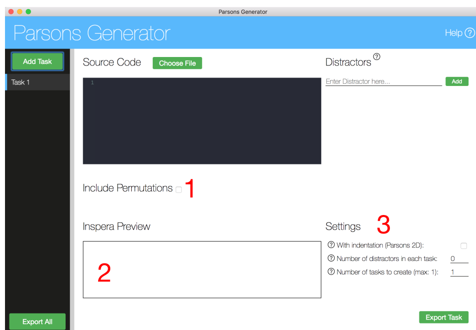

Figure B.4: Application with an empty task (Bottom of the task page)

B.1.3 Edit Task

Add Task Title

At the top of the task page, there is placed an input field to set the task title (refer to 1 in Figure B.3).

If a task title is set, this will be set as the task title in Inspera as well. If the task title is omitted, an UUID (Universal Unique Identifier) is generated and set as task title in Inspera.

Add Task Description

A description can be added to the task with the rich text editor (refer to 3 in Figure B.3). The description can be written in Bokmål, English, and/or Nynorsk with the help of the tab system for the description (refer to 2 in Figure B.3).

NB! It is possible to paste in formatted text to the task description, but this might cause errors when exporting the task. We advice you not to paste formatted text, since this might cause problems with Inspera.

Upload File

To upload a code snippet to the application, click the 'Choose File' button (refer to 4 in Figure B.3).

Source Code Editor

To manually type a code snippet in the application, use the source code editor (refer to 5 in Figure B.3).

Add Distractors

To add distractors, use the input field to the right of the source code editor (refer to 6 in Figure B.3).

Distractors are incorrect alternatives to the solution, which sole purpose is to distract the user. There is no upper limit to how many distractors that can be added.

To delete a distractor, hover the item in the list of distractors, and click the red 'x' that appears.

Include Permutations

To include permutations for the task, toggle the 'Include Permutations' checkbox (refer to 1 in Figure B.4).

'Include Permutations' refers to finding all correct permutations of the given code lines. This is beneficial for discovering dependencies in the code snippet, and create tasks that have more than one solution.

To find all permutations, a Directed Acyclic Graph (DAG) has to be provided. This is done by stating which code lines must come before other code lines (refer to 2 and 3 in Figure B.5). For each line in the dropdown menu, state which lines must come **after** this line by checking the checkboxes.
For example, if Line 1 has to come before Line 3 (that is, Line 3 depends on Line 1), then you select Line 1 in the dropdown menu (refer to 2 in Figure B.5) and then check the checkbox for Line 3 in the list below (refer to 3 in Figure B.5). **NB:** If Line 1 must come before Line 3, and this is stated in the application, the opposite pairing (Line 3 must come before Line 1) is disabled.

Because of the time effort in finding permutations, an 'Update Permutations' button has to be clicked whenever the permutations have been altered (refer to 1 in Figure B.5).

To the right of the DAG creation section, a Permutations Preview section is displayed (refer to 4 and 5 in Figure B.5). This section gives a preview of all the permutations that were found, number of false positives, and the probability of guessing a correct solution based on luck.

Using the preview box (refer to 5 in Figure B.5), one can navigate through all the permutations found, and see if everything is correct.
In some cases, there will be generated false positives, which are solutions not logically correct, but which Inspera will deem correct anyway. These will be highlighted with a red warning in the preview box (refer to 1 in Figure B.6).

Permutations are only supported for normal Parsons problems, and not 2D Parsons problems. For this reason, if the 'Include Permutations' checkbox is on while a 2D Parsons task is being created (refer to 1 and 2 in Figure B.7), an error message will be displayed in the 'Include Permutations' section.

Inspera Preview

A preview of how the task will look like in Inspera is shown at the bottom of the task page (refer to 2 in Figure B.4 and 1 in Figure B.9).

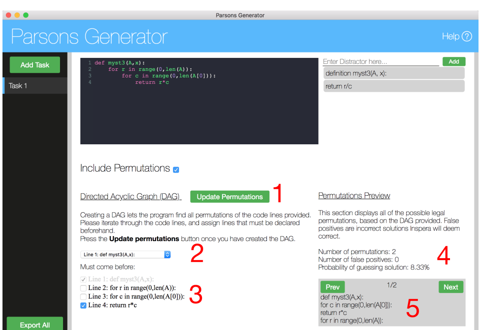

Figure B.5: Permutations Settings for a task

Figure B.6: Permutations w/ False Positives

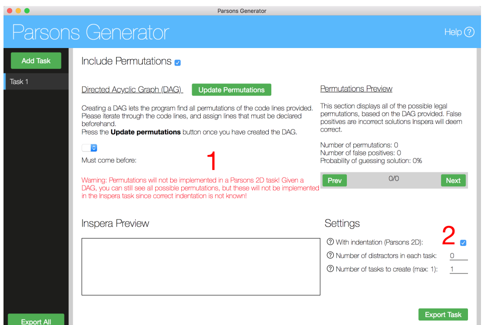

Figure B.7: Permutation warning

Settings

The task has three settings at the bottom of the task page (refer to 3 in Figure B.4):

1. 'With indentation (Parsons 2D)' decides whether the task should be a standard Parsons problem or a 2D Parsons problem (refer to 1 and 2 in Figure B.8 and Figure B.9).
2. 'Number of distractors in each task' sets the number of distractors to include in the task. The distractors are chosen from the pool of distractors added earlier in the task (refer to 6 in Figure B.3).
3. 'Number of tasks to create (max: #)' sets the number of unique tasks to generate. The uniqueness comes from each task having a unique subset of distractors, and thus there is an upper limit on the number of unique tasks possible to create.

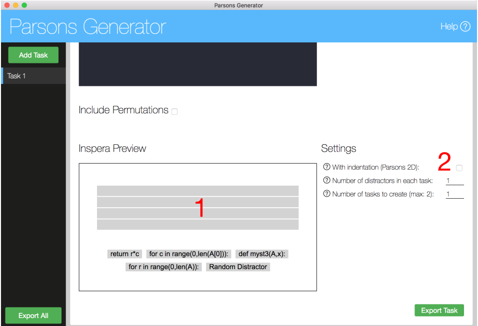

Figure B.8: Normal Parsons Problem

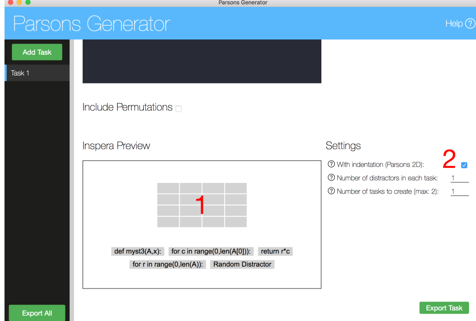

Figure B.9: Parsons 2D Problem

B.1.4 Delete Task

To delete a task, hover the task in the task list at left hand side of the application, and a red button with an 'x' will appear behind the task title (refer to 1 in Figure B.10). Click this button to delete the task.

If the task to be deleted is currently displayed, the user will be redirected to the first task in the task list upon deleting the current task.

B.1.5 Export Task(s)

To export a single task, click the button 'Export Task' at the bottom of the task page (refer to 1 in Figure B.11). A file dialog will appear, and let you name the file

and decide export destination.

To export a set of multiple tasks, click the button ’Export All’ at the bottom of the task list (refer to 2 in Figure B.11). A file dialog will appear, and let you name the file and decide export destination.

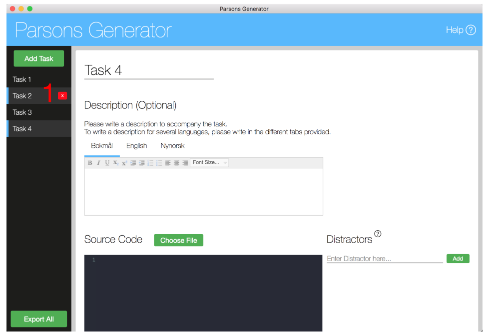

Figure B.10: Delete Task

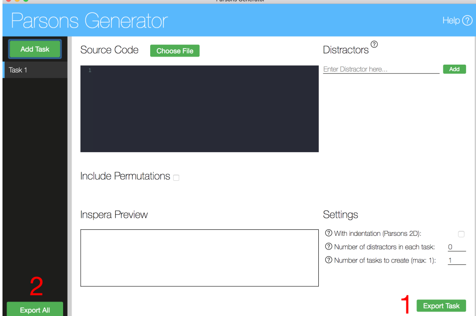

Figure B.11: Export Task(s)

B.1.6 Helper Icons

Throughout the application there are helper icons (question mark inside a circle) that will display help dialogs on hover (refer to 1 in Figure B.12).

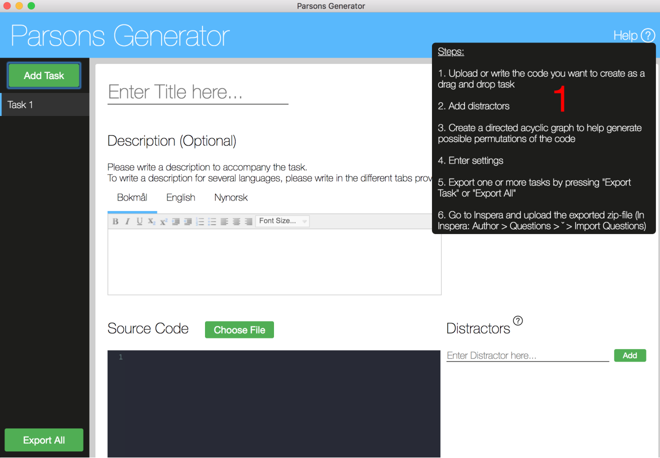

Figure B.12: Helper Functionality

B.2 Uploading to Inspera

Once you have exported one or more tasks, they are ready to be uploaded to Inspera. Inspera provides a tutorial for this process on their web site: https://inspera.atlassian.net/wiki/spaces/KB/pages/153813089/QTI+2.1+Export+and+Import+questions (Accessed: 08/05/19)

Appendix C

Use Cases

C.1 Iteration 1

The use cases for iteration 1 were as follows:

Steps of Execution	Expected Result
1. Execute QTI Converter manually through terminal (which includes hard coded path to a Python file)	Generate folder with XML content based on Python file
1. Execute QTI Converter manually through terminal (which includes hard coded path to a Python file) 2. Compress generated folder (create a zip-file) 3. Upload compressed folder to Inspera	Get a functional task in Inspera with the same code lines as the initial Python file

1. Open the terminal 2. Navigate to the source folder of the project 3. Execute 'npm start'	An empty Electron application should open
1. Open the terminal 2. Navigate to the source folder of the project 3. Execute 'flask run'	Flask should run on http://127.0.0.1:5000/, which should be confirmed in the terminal

C.2 Iteration 2

For all of the use cases regarding the user interface, it is assumed that the application is already running. The use cases performed and passed in this iteration were:

Steps of Execution	**Expected Result**
1. Type source code (or any text) in the source code editor	Internal state of the application should be updated (can be checked with console.log())
1. Copy source code (or any text) and paste it in the source code editor	Internal state of the application should be updated (can be checked with console.log())
1. Click the 'Upload File' button 2. Select and open a code file (or any other text file)	Internal state of the application should be updated, and the source code editor should represent the contents of the opened file

1. Populate the source code editor with source code (or any other text) 2. Select file destination 3. Press the 'Download' button	The application should generate a zip file at given destination, and the server should respond with a confirmation (in the GUI)
1. Populate the source code editor with source code (or any other text) 2. Export task and upload to Inspera	The drag areas should be placed in a random order
1. Populate the source code editor with source code that has indentations 2. Check 2D Parsons checkbox 3. Export task and upload to Inspera	The drop areas should have multiple columns corresponding to number of max indentations (Drop area grid). Code lines should connected to correct row and column
1. Populate the source code editor with source code that has indentations 2. Uncheck 2D Parsons checkbox 3. Export task and upload to Inspera	The drop areas should have one single column. Code lines should be connected to correct row

1. Package the application using macOS (create a build version for macOS) 2. Package the application using Windows (create a build version for Windows) 3. Run the executable application file on macOS 4. Run the executable application file on Windows	The application should be executable on both macOS and Windows, with all current features included

C.3 Iteration 3

For all of the use cases regarding the user interface, it is assumed that the application is already running, and for each use case the application is reset. The use cases performed and passed in this iteration were:

Steps of Execution	Expected Result
1. Click the button 'Add Task'	An empty task should appear in the application
1. Click the button 'Add Task' twice	An empty task should appear in the application. Additionally, there should be two tasks in the list of tasks, where the second task is currently selected and displayed
1. Click the button 'Add Task' twice 2. Click on the first task in the list of tasks	The first task should be displayed

1. Click the button ’Add Task’ 2. Edit the task title to ’Test task 1’	In the task list, ’Task 1’ should be updated to ’Test task 1’
1. Click the button ’Add Task’ twice 2. Edit the current tasks title to ’Test task 2’ 3. Click on the first task in the list of tasks 4. Edit the current tasks title to ’Test task 1’ 5. Got back to the second task (by clicking on ’Test task 2’ in the list of tasks)	Task 2 should be displayed with a title of ’Test task 2’
1. Click the button ’Add Task’ twice 2. Add code to source code editor in ’Task 1’ 3. Change tabs to ’Task 2’ 4. Add code to source code editor in ’Task 2’ 5. Change tabs to ’Task 1’	’Task 1’ should still display the code added for this task. Correct internal state management and addition can also be controlled with console.log of internal state

1. Create multiple tasks 2. Change between tabs and make changes multiple times	All changes made should be stored in the internal state for the correct task object
1. Create multiple tasks with different code 2. Press 'Export All' button 3. Upload zip-file to Inspera	All created tasks should appear in the same question set with the correct data

C.4 Iteration 4

For all of the use cases regarding the user interface, it is assumed that the application is already running, and for each use case the application is reset. The use cases performed and passed in this iteration were:

Steps of Execution	**Expected Result**
1. Click the button 'Add Task' 2. Click the button 'Delete Task'	The task should be deleted, rendering the task list and task page empty
1. Click the button 'Add Task' 2. Add the title 'Use case test 1' to the task 3. Click the button 'Delete Task' 4. Click the button 'Add Task' again	An empty task page should be displayed in the application (with no title set)

1. Click the button 'Add Task' twice 2. Click the button 'Delete Task'	The second task should be deleted, and the task page should update to display Task 1. The task list should now only contain Task 1
1. Click the button 'Add Task' 2. Write a distractor in the distractor input field 3. Click the 'Add' button next to said input field	The distractor should be added to a list of distractors below the input field
1. Click the button 'Add Task' 2. Write a distractor in the distractor input field 3. Press the Enter key on the keyboard	The distractor should be added to a list of distractors below the input field
1. Click the button 'Add Task' 2. Write a distractor in the distractor input field 3. Click the 'Add' button next to said input field 4. Write another distractor in the distractor input field 5. Click the 'Add' button next to said input field	Both distractors should be added to the list of distractors below the input field

<table>
<tr><td>1. Click the button ’Add Task’
2. Write a distractor in the distractor input field
3. Click the ’Add’ button next to said input field
4. Hover the distractor in the list of distractors, and press the red button containing an ’x’</td><td>The distractor list should be rendered empty when the distractor is deleted</td></tr>
<tr><td>1. Click the button ’Add Task’
2. Write a distractor in the distractor input field
3. Click the ’Add’ button next to said input field
4. Write another distractor in the distractor input field
5. Click the ’Add’ button next to said input field
6. Hover the first distractor in the list of distractors, and press the red button containing an ’x’</td><td>The list of distractors should now contain only the second distractor</td></tr>
<tr><td>1. Click the button ’Add Task’
2. Add multiple distractors
3. Delete some of the distractors</td><td>The removed distractors should also be removed from the internal state</td></tr>
</table>

1. Click the button 'Add Task' 2. Click the button 'Export Task'	The application should display a file dialog, where it is possible to name the exported file
1. Click the button 'Add Task' 2. Click the button 'Export Task' 3. Choose export destination	The application should display a confirmation message saying that the task was successfully exported
1. Click the button 'Add Task' 2. Hover the help button on the top right of the application	On hover, it should appear helper text describing how to use the application
1. Click the button 'Add Task' 2. Write code in the source code editor 3. Add multiple distractors 4. Export and upload task to Inspera	All the given distractors should be their own drag areas placed in a random order together with the other lines of code. The distractor drag areas should not be connected to any drop area

C.5 Iteration 5

For all of the use cases regarding the user interface, it is assumed that the application is already running, and for each use case the application is reset. The use cases performed and passed in this iteration were:

Steps of Execution	Expected Result

1. Click the button ’Add Task’ three times 2. Hover the second task in the task list, and press the red box with an ’x’ (Delete the task)	The second task should be deleted. The task list should now contain two elements: Task 1 and Task 3
1. Click the button ’Add Task’ three times 2. Go to task 2 3. Hover the second task in the task list, and press the red box with an ’x’ (Delete the task)	The second task should be deleted, and the task page should update to display Task 1. The task list should now contain two elements: Task 1 and Task 3
1. Click the button ’Add Task’ 2. Paste highly formatted text into the task description 3. Click the button ’Export Task’ 4. Choose export destination	The application should display an error message saying that the formatted description is not supported by Inspera
1. Click the button ’Add Task’ 2. Write text in task description for all languages 3. Export and upload to Inspera	The Inspera task should display the same task description in the same format as written in the IT artefact for the correct languages

1. Click the button 'Add Task' 2. Enable permutation support	Display the DAG creation (dropdown menu with checkboxes), permutation options and permutation preview. Also display a warning message if 2D Parsons is checked
1. Click the button 'Add Task' 2. Write code in source code editor 3. Enable permutation support 4. Create a DAG for the code 5. Click the 'Update permutations' button	See the number of correct permutations as well as the number of false positives. All these permutations should also be displayed in the permutations preview
1. Click the button 'Add Task' 2. Write code in source code editor 3. Enable permutation support 4. Create a DAG for the code 5. Click the 'Update permutations' button 6. Export and upload to Inspera	All permutations previewed in the IT artefact should be represented in Inspera by the use of multiple connections between drag and drop areas

1. Click the button ’Add Task’ 2. Write over 10 lines of code in source code editor 3. Enable permutation support 4. Do not fill out any information in the DAG 5. Click the ’Update permutations’ button	See a warning message letting the user know there exists too many permutations and that the DAG needs additional information or the task itself should be changed
1. Click the button ’Add Task’ 2. Add y number of distractors 3. Set ’Number of distractors in each task’ to x	The max number of tasks to create should be set to the binomial coefficient of x over y. The user should not be able to set the ’Number of tasks to create’ to any higher number than the binomial coefficient and no lower than 1
1. Click the button ’Add Task’ 2. Add 5 distractors 3. Set ’Number of distractors in each task’ to 2 4. Set ’Number of tasks to create’ to 10 5. Export and upload to Inspera	The generated question set should consist of 10 tasks that all have two different distractors implemented

1. Click the button 'Add Task' 2. Add code to source code editor 3. Make sure two or more lines of code are identical (identation levels does not matter, just the text) 4. Export and upload to Inspera	All equal lines of code should be connected to the exact same drop areas

Appendix D

Requirements

D.1 List of Requirements

ID	Title	Description	Iteration added
1	Desktop Application	The IT artefact should be a desktop application providing the users with a GUI where one can perform all necessary actions to properly generate Parson problems	1
2	Parsons Problems and 2D Parson problems	The IT artefact has to support the generation of both unmodified Parson problems as well as 2D Parson problems	1
3	Upload file	Users should be able to upload any code file to the IT artefact, such that this code file can be used as the basis for the Parson problem to be generated	1
4	Copy and paste source code	Users should be able to copy and paste source code into the IT artefact and use this source code as the basis for the Parson problem to be generated	1
5	Write and edit source code	Users should be able to write and edit source code in the IT artefact and use this source code as the basis for the Parson problem to be generated	1

6	Generate Inspera supported QTI	The Parson problem to be generated by the IT artefact must be generated in a QTI format that is supported by Inspera	1
7	Parsons problems design	The Parsons problems to be generated by the IT artefact must follow the predetermined design of how Parsons problems should be created in Inspera using drag and drop tasks	1
8	Support distractors	The user should be able to add distractors to the Parson problems	1
9	Generate multiple unique tasks (w/ different distractors)	The IT artefact should be able to use a subset of the given distractors to generate multiple unique tasks (of the same algorithm) with different distractors	1
10	Generate multiple tasks	The IT artefact should be able to generate multiple tasks (of different algorithms) at the same time. It should be possible to create, edit and delete these tasks	1
11	Export all and Export single	When creating multiple tasks, the user should be able to export both a single task and all tasks	1
12	Task description in English, Norwegian and Nynorsk	The IT artefact should support the creation of a task description in English, Norwegian and Nynorsk. (This can be done in Inspera itself rather than in the IT artifact, but might be problematic if many tasks are to be generated)	1
13	Preview	The IT artefact should show the user a preview of the task to be generated so potential errors are discovered as early as possible (before exporting and uploading the task to Inspera)	2
14	Export destination	The user should be able to select the exact folder the exported task should be placed in when exporting from the IT artefact	2
15	Help functions	The IT artefact should provide the user with informative and detailed help functions to clearly communicate what features do and how things work	2

16	Tutorial	The IT artefact should provide the user with proper tutorial showing the workflow, how things work, and why the IT artefact is useful	4
17	Confirmation, warning and error messages	The IT artefact should provide the user with a proper confirmation, warning, and error messages. For example, when deleting a task, a confirmation box should be displayed to avoid extra work if the button was pressed by accident	4
18	Persistent storage	The IT artefact should support persistent storage of the tasks so a user can save, quit and resume a session	4
19	Permutations	The IT artefact should support code permutations to avoid incorrect grading if students find other correct rearrangements of the code. The generated task must make sure that all correct permutations are properly represented. The IT artefact should also provide proper warnings if there exists any permutations that count as false positives	4

D.2 Non-functional Requirements

ID	**Title**	**Description**	**Iteration added**
1	Performance	All actions should have a response time of less than 2 seconds	1
2	Availability	The IT artefact should be available on Windows and MacOS	1
3	Security	The IT artefact has to make sure all created tasks are as safe as any other exam question created by a user on their computer	1
4	Usability	The usability of the IT artefact in regards to efficiency, effectiveness, and satisfaction should be better than the creating of the same tasks in Inspera	1

5	Reliability	The IT artefact should never generate faulty tasks without proper warnings and error messages. A user should not have to double check the generated task in Inspera to see if everything is correct	1
6	Ease of use	The IT artefact should be as intuitive to use, understand, and learn as possible. One can also assume some given domain knowledge given the expected users of the system. The IT artefact should also have focus on design, aesthetics and UX to improve ease of use and satisfaction	1
7	Feedback	The IT artefact should provide the user with clear and detailed feedback and information regarding the consequences of each action	1
8	Maintenance	The written code should be well documented to support future maintenance and development	1
9	Inspera support features	The Parson problems to be generated can only consist of features already supported in Inspera. Adding any additional features in the QTI that is not supported by Inspera is against their uploading rules	1

D.3 Removed Requirements

ID	Title	Description	Iteration added / removed
1	Suggest distractors	The IT artefact should suggest distractors that seem fitting to the given task	1 / 2
2	Comment syntax	Instead of having to use a GUI, the user should be able to comment source code with a specific syntax and achieve the same functionality as they would with the GUI	1 / 2

3	Score allocation	The IT artefact should support different score allocations for each line of code in a task, so a user can give the most difficult parts of the algorithm more allocated points than the easier parts	1 / 2

D.4 List of Improvements Discovered during User Testing

Title	Description
Help-functions are good, but can still be improved	Most participants used the help-functions multiple times to better understand the application, but many still did not understand what 'With indenting' and 'Distractors' ment. The general flow and relation between the app and Inspera was also unclear even after using the help-functions
Trouble with file-chooser	1) File-chooser can get hidden behind the application 2) One can open multiple file-choosers, which makes them all nonfunctional 3) System crash after a task is deleted and one tries to upload a new file with 'Choose file' 4) When the zip-file is exported, it's missing a "filename already exists. Do you want to overwrite?"-popup if the name already exists
'With indenting'-toggle is not intuitive and not clearly understood	1) Not clear what its purpose 2) Expects something to happen when you press the toggle-button. Currently, no feedback is given when pressed 3) Easy to overlook or forget. Should be more clear and visible 4) It is unclear that this function only affects the generated Inspera task, and nothing in the app itself 5) Text, toggle-box, and help-function does not seem to be connected or related (Too far apart) 6) The help-function did not necessarily give a better understanding of its purpose

Uses 'Add'-button before filling out input-field	The input-field is currently placed on the right side of the 'Add'-button, something that goes agains the 'left-to-right flow' a user typically has. Many pressed the 'Add'-button first and expected an input-field to appear. At the moment, this is not what happens. The user is expected to type in their distractor in the input-field first, and then press 'Add'
Unclear/vague input-fields	The title and distractor input-fields are somewhat vague (some participants thought they were headers)
Uncertainty around the name 'Distractors'	Not everyone knows what a 'Distractor' is (or means)
Uncertainty and doubt around Parsons Problems as a task type in Inspera	Parsons Problems in Inspera can be difficult for students to complete, and thus, might not be suited for tests and exams in many cases
Does not like the code editor theme	Many are not used to the chosen code editor theme and would like the possibility to choose their own preferred editor theme. Some also mentioned that there was not much to be gained by improving the code editor (with multiple themes, shortcuts, compilation etc.) since most users would want to write all the code in their own text editor, and not in the application
Width of drag-areas generated by the application should be minimal	Since the text in each drag-area is centered, the width of the drag-area has to be exactly the length of the text to avoid extra spacing within a drag-area. This kind of spacing might look like indentation when working with code. The generated drag-areas has some inaccuracies regarding the width of the area compared to the text itself. By using specific font-metrics one should be able to calculate the exact length of each line of code

'Delete task'-button is missing coherence	'Delete task'-button is placed at the top right corner together with the 'Export task'-button. This placement makes it look like the two buttons are connected and has some kind of coherence, when in reality, they dont. An option was to place the 'Delete task'-buttons in the task-overview in the left margin (maybe as an X in each task in the list)
Unexpected placement of 'Export task'-button	The 'Export task'-button is placed at the top right corner, but should be at the bottom right corner since this is the final step a user performs in the application
Uncertainty regarding the link between the application and Inspera	The connection and flow between the application, the zip-file, and Inspera is not necessarily intuitive. What happens after one presses the 'Export task'-button is also unclear. One should show how everything is connected and how the application actually contributes to the process. There was also some uncertainty around exactly when the user was done with the application and was supposed to start using Inspera
Missing tutorial	The connection and flow between the application, the zip-file, and Inspera is not necessarily intuitive. The usefulness of the application is not too clear either if one has not already used Inspera to create drag and drop tasks. A tutorial could help showing how things actually work and how the application is useful
Usefulness not intuitive	The usefulness of the application isnt too clear either if one has not already used Inspera to create drag and drop tasks and knows how time-intensive the process is. Without this knowledge, it was difficult to know what the application did and why. One should try to better show how the application is connected with Inspera and what it actually does

Wants copy paste functionality in the application	Copy pasting in the application did not work during testing (due to a small bug), and this is both expected and should actually work
Wants a preview	To better understand and see what the application does, a preview could solve a lot of the uncertainty. Actually having to upload the task to Inspera before one sees a result was not optimal. With a preview one could also better see what the 'With indenting' function does, see how the grid layout will become and choose their own drop-area width
Wants zoom functionality	For those with larger screens, the application and the text became quite small. Some tried to zoom in on the application to solve this, but zooming is not supported
Wants a confirmation box before deleting a task	When one presses the 'Delete task'-button, the task instantly disappears. There should be a confirmation box before it is deleted to avoid extra work if the button was pressed by accident
Wants to edit distractors	The distracors are added to a list, and multiple participants tried to press the list item in order to edit the distractor. This is currently not supported by the application
Wants the option to use a script or the command-line	Since the GUI does not have too much functionality (at the time of the user testing), some would like to use the command-line instead of the GUI. Both the GUI and a command-line should work, so the user can choose between them both
Wants the possibility to upload images in the application	Some would like to generate their own images with special characters, formulas, or fonts. They would then want to upload these images to the application to generate drag and drop tasks with these images instead of code

Wants the possibility to use their own editor for both the code and the distractors.	A normal workflow consisted of the user creating both the code and the distractors in their own text editor, and therefore wanted to copy paste or upload everything directly to the application. A simpler way to add distractors was preferable. Filling in one by one was a tedious task for some. Creating their own "distractor syntax" could also let users write the distractors directly in the code, and let the application automatically extract all distractor once the code is uploaded
Wants support for special characters in the editor (Sigma, delta, etc.)	By supporting latex in the editor, the user can create tasks with special characters
Some sort of persistence of tasks. Downloaded zip-file could potentially be able to be uploaded to the application for further editing (instead of other kinds of storage functionality)	The application should probably support some kind of persistence, so that it is possible to edit and work on the tasks sometime later without creating them from scratch again. One kind of persistence could be uploading the generated zip file to the application again. It was also mentioned that, since the export format is somewhat impractical (Inspera specific), it might be a good idea to define a persistence format and an explicit export (and import))
Want to be able to generate other task types as well	Being able to generate other types of tasks than drag- and drop tasks would be nice. This is because some people are not sure of the legitimacy of Parsons Problems as a type of task in Inspera
Option to set the size of drop areas	Would be nice to be able to define the width of each drop area, and the distance between each area.

Users might want to type task description in the application	Would be nice to create the entire task in the application, instead of doing the leftover work after generating the task. This has to support Norwegian, 'Nynorsk' and English
It's called "indentation", not "Indenting"	Grammatical error
Option to choose toggle between Norwegian and English	Would like to toggle between the language used (for the task) in the application
Support permutation	In an algorithm there are many cases where some lines of code can be placed in different positions and still work. If the app does not support such rearranged positions where the code would still work, the tasks may result in incorrect grading
Different tabs	If different tab standards (2 spaces, 4 spaces, 1 tab) are used in the same algorithm, the task generator will not understand how to properly add indentation. Providing a warning or some check for this could help

Appendix E

Transcription Categories

E.1 Inspera

E.1.1 Positive Categories

ID	Category	Frequency
01	Bruker forhåndsvisning	1

E.1.2 Negative Categories

ID	Category	Frequency
02	Generell irritasjon/frustrasjon på Inspera (ikke intuitivt)	21
03	Vanskelig å manipulere posisjon, høyde og bredde til dra- og slippområder	9
04	Ikke intuitivt hvordan man kobler sammen dra og slippområde	8
05	Irritasjon (default plassering av nye felter)	6
06	Misfornøyd med sluttresultatet (estetisk sett)	6
07	Prøver å forstå Inspera	5
08	Vil skrive tekst direkte i editoren, ikke på høyresiden	4
09	Ønsker copy paste av dra- og slippområder	3
10	Bruker mangler forventet funksjonalitet i løsningen	2
11	Hvordan forstørre oppgavefeltet?	2
12	Bruker finner ikke ønsket funksjonalitet	2
13	Vil heller bruke LaTeX	1
14	Ønsker å skrive distraktorer i egen editor	1
15	Ønsker left align av tekst i draområder	1
16	Ønsker box alignment i Inspera	1

E.2 IT artefact

E.2.1 Positive Categories

ID	Category	Frequency
17	Ser nytteverdi av appen	7
18	Utnytter / liker hjelpefunskjon	7
19	Add task og Choose file er intuitiv	6
20	Lett å lære	5
21	Liker ’Enter’-knapp for å legge til distraktor	5
22	Export all er intuitiv	3
23	Export task er intuitiv	1
24	Forstår sammenkobling mellom appen, zip og inspera	1
25	Fornøyd med GUI	1

E.2.2 Negative Categories

ID	Category	Frequency
26	'With indenting'-toggle er utydelig og lite intuitiv	11
27	Ønsker copy paste i appen	9
28	Bruker 'Add'-knapp før input-felt	8
29	Ønsker å ha mulighet til å bruke egen editor til både kildekode og distraktorer	6
30	Usikkerhet rundt kobling mellom app og Inspera	4
31	Ønsker forhåndsvisning	4
32	Trøbbel med filbehandler	3
33	Utydelig input-felt	3
34	Usikker på Parsons Problems som oppgavetype	3
35	Liker ikke editor theme	3
36	Delete task mangler samhørighet	3
37	Usikkerhet rundt navnet distraktor	2
38	Bredde på draområder generert av appen bør være minimal	2
39	Nytteverdi ikke intuitiv	2
40	Ønsker confirmation box når en sletter task	2
41	Ønsker å edite distraktor	2
42	Ønsker å ha mulighet til å bruke script/kommandolinje	2
43	Ønsker å ha muligheten til å laste opp bilder i appen	2
44	Uventet plassering av 'Export'-knapp	1
45	Mangler tutorial	1
46	Ønsker zoom funksjonalitet	1
47	Ønske for støtte av spesielle tegn i editoren (Sigma, delta etc)	1
48	Nedlastet zip fil bør kunne lastes opp i appen og endres videre (i stedet for annen storage funksjonalitet)	1
49	Ønske om å kunne generere andre oppgavetyper også	1
50	Vil selv velge størrelse på slippområder	1
51	Brukere vil kanskje skrive oppgavetekst i applikasjonen	1

Appendix F

The Keystroke-Level Model for User Performance Time

F.1 The Keystroke-Level Model using Inspera

Calculating user performance time using the keystroke-level model (Card et al., 1980). Averaged skilled typist (55 wpm) is assumed in the following calculations.

F.1.1 Adding Drop Areas

Step	Description	Operator	Duration (sec)
1	Mentally prepare for getting started by Heuristic Rule 0	M	1.35
2	Move cursor to 'Drop areas' dropdown (no M by Heuristic Rule 1)	P	1.1
3	Click mouse button (no M by Heuristic Rule 0)	K	0.2
4	Mentally prepare for adding a drop area field	M	1.35
5	Move cursor to 'Add drop area' button (no M by Heuristic Rule 1)	P	1.1
6	Click mouse button (no M by Heuristic Rule 0)	K	0.2
7	Mentally prepare for replacing new drop area	M	1.35

8	Move cursor to new drop area (no M by Heuristic Rule 1)	P	1.1
9	Click mouse button (no M by Heuristic Rule 0)	K	0.2
10	Drag cursor to new correct placement of drop area (no M by Heuristic Rule 1)	P	1.1
11	Release mouse button (no M by Heuristic Rule 0)	K	0.2
12	Mentally prepare for resizing of new drop area	M	1.35
13	Move cursor to corner of drop area (no M by Heuristic Rule 1)	P	1.1
14	Click mouse button (no M by Heuristic Rule 0)	K	0.2
15	Move cursor correct resize position (no M by Heuristic Rule 1)	P	1.1
16	Release mouse button (no M by Heuristic Rule 0)	K	0.2

Creating first drop area (Step 1-16)
$4*M+6*P+6*K = 4*1.35+6*1.1+6*0.2 = 5.4+6.6+1.2 = 13.2 seconds$

Adding subsequent drop areas (Repeat step 4-16 for every new drop area)
$3*M+5*P+5*K = 3*1.35+5*1.1+5*0.2 = 4.05+5.5+1 = 10.55 seconds$

F.1.2 Adding Drag Areas

The average word count of each line of code in Task 1 and Task 2 is 14.

Step	Description	Operator	Duration (sec)
1	Mentally prepare for getting started by Heuristic Rule 0	M	1.35
2	Move cursor to 'Drag areas' dropdown (no M by Heuristic Rule 1)	P	1.1
3	Click mouse button (no M by Heuristic Rule 0)	K	0.2
4	Mentally prepare for adding new drop area by Heuristic Rule 0	M	1.35
5	Move cursor to 'Add drag area' button (no M by Heuristic Rule 1)	P	1.1
6	Click mouse button (no M by Heuristic Rule 0)	K	0.2
7	Mentally prepare for writing drag area text by Heuristic Rule 0	M	1.35
8	Move cursor to new drag area input field (no M by Heuristic Rule 1)	P	1.1
9	Click mouse button (no M by Heuristic Rule 0)	K	0.2
10	Write new 14-letter word (Average word count of Task1 and Task2)(no M by Heuristic Rule 0)	14K	2.8
11	Mentally prepare for replacing the new drag area Heuristic Rule 0	M	1.35
12	Move cursor to new drag area (no M by Heuristic Rule 1)	P	1.1
13	Click mouse button (no M by Heuristic Rule 0)	K	0.2
14	Drag cursor to new correct placement of drag area (no M by Heuristic Rule 1)	P	1.1
15	Release mouse button (no M by Heuristic Rule 0)	K	0.2
16	Mentally prepare for resizing of new drag area	M	1.35

17	Move cursor to corner of drag area (no M by Heuristic Rule 1)	P	1.1
18	Click mouse button (no M by Heuristic Rule 0)	K	0.2
19	Move cursor correct resize position (no M by Heuristic Rule 1)	P	1.1
20	Release mouse button (no M by Heuristic Rule 0)	K	0.2
21	Mentally prepare for connecting drag area to correct drop area	M	1.35
22	Move cursor to drag area (no M by Heuristic Rule 1)	P	1.1
23	Click mouse button (no M by Heuristic Rule 0)	K	0.2
24	Move cursor to drop area (no M by Heuristic Rule 1)	P	1.1
25	Click mouse button (no M by Heuristic Rule 0)	K	0.2

Creating first drag area (Step 1-25)
$6*M+9*P+23*K = 6*1.35+9*1.1+19*0.2 = 8.1+9.9+3.8 = 22.6 seconds$

Adding subsequent drag areas (Repeat step 4-25 for every new drag area)
$5*M + 8*P + 22*K = 5*1.35 + 8*1.1 + 18*0.2 = 6.75 + 8.8 + 3.6 = 19.95 seconds$

Adding subsequent distractor drag areas (Repeat step 4-20 for every new distractor)
$4*M+6*P+20*K = 4*1.35+6*1.1+20*0.2 = 5.4+6.6+4 = 16 seconds$

Task 1 has 8 lines of code to add. 4 correct lines of code and 4 distractors, and a total of 4 indentation levels since it was to be created as a 2D Parsons problem. Task 1 thus required a 4*4=16 drop area grid.
Task 2 has 12 lines of code to add. 7 correct lines of code and 5 distractors, and a total of 1 indentation level since it was to be created as a normal Parsons problem. Task 2 thus required a 7*1=7 drop area grid.

Total time required for Task 1
(Step 1-16) + 15*(Step 4-16) + (Step 1-25) + 3*(Step 4-25) + 4*(Step 4-20) = 13.2 + 15*10.55 + 22.6 + 3*19.95 + 4*16 = 317.9 seconds

Total time required for Task 2

(Step 1-16) + 6*(Step 4-16) + (Step 1-25) + 6*(Step 4-25) + 5*(Step 4-20) = 13.2 + 6*10.55 + 22.6 + 6*19.95 + 5*16 = 298.8 seconds

F.2 The Keystroke-Level Model using IT Artefact

Calculating user performance time using the keystroke-level model (Card et al., 1980). Averaged skilled typist (55 wpm) is assumed in the following calculations.

F.2.1 Adding a Task

Step	Description	Operator	Duration (sec)
1	Mentally prepare for getting started by Heuristic Rule 0	M	1.35
2	Move cursor to 'Add Task' button (no M by Heuristic Rule 1)	P	1.1
3	Click mouse button (no M by Heuristic Rule 0)	K	0.2
4	Mentally prepare for writing code by Heuristic Rule 0	M	1.35
5	Move cursor to code editor (no M by Heuristic Rule 1)	P	1.1
6	Click mouse button (no M by Heuristic Rule 0)	K	0.2
7	Mentally prepare for writing new word by Heuristic Rule 0	M	1.35
8	Write new 14-letter word (Average word count of Task1 and Task2)(no M by Heuristic Rule 0)	14K	2.8
9	Mentally prepare for writing distractors by Heuristic Rule 0	M	1.35
10	Move cursor to distractor input field (no M by Heuristic Rule 1)	P	1.1
11	Click mouse button (no M by Heuristic Rule 0)	K	0.2
12	Write new 14-letter word (Average word count of Task1 and Task2)(no M by Heuristic Rule 0)	14K	2.8
13	Click enter to add distractor (no M by Heuristic Rule 0)	K	0.2
14	Mentally prepare for choosing 2D Parsons or not by Heuristic Rule 0	M	1.35
15	Move cursor to 2D Parsons checkbox (no M by Heuristic Rule 1)	P	1.1

16	Click mouse button (no M by Heuristic Rule 0)	K	0.2
17	Mentally prepare for exporting task by Heuristic Rule 0	M	1.35
18	Move cursor to 'Export Task' button (no M by Heuristic Rule 1)	P	1.1
19	Click mouse button (no M by Heuristic Rule 0)	K	0.2
20	Mentally prepare for deciding folder destination by Heuristic Rule 0	M	1.35
21	Move cursor to file chooser window (no M by Heuristic Rule 1)	P	1.1
22	Click mouse button (no M by Heuristic Rule 0)	K	0.2
23	Move cursor to 'Save' button (no M by Heuristic Rule 1)	P	1.1
24	Click mouse button (no M by Heuristic Rule 0)	K	0.2
25	Mentally prepare for confirmation message by Heuristic Rule 0	M	1.35
26	Move cursor to confirmation message (no M by Heuristic Rule 1)	P	1.1
27	Click mouse button (no M by Heuristic Rule 0)	K	0.2

Task 1 has 8 lines of code to add. 4 correct lines of code and 4 distractors.
Task 2 has 12 lines of code to add. 7 correct lines of code and 5 distractors.

Step 7 must be repeated for every line of code to add
Step 8-12 must be repeated for every distractor to add

Total time required for Task 1
(Step 1-6) + 4*(Step 7-8) + 4*(Step 9-13) + (Step 14-27) = 5.3 + 4*4.15 + 4*5.65 + 10.55 = 55.05 seconds

Total time required for Task 2
(Step 1-6) + 7*(Step 7-8) + 5*(Step 9-13) + (Step 14-27) = 5.3 + 7*4.15 + 5*5.65 + 10.55 = 73.15 seconds

Appendix G

Declaration of Consent

Vil du delta i forskningsprosjektet

"Effektiv generering av Parsons problems for digitale programmerings-eksamener i Inspera"?

Dette er et spørsmål til deg om å delta i et forskningsprosjekt hvor formålet er å evaluere effekten av et program som automatisk genererer dra-og-slipp oppgaver i Inspera, framfor å manuelt utvikle slike oppgaver. I dette skrivet gir vi deg informasjon om målene for prosjektet og hva deltakelse vil innebære for deg.

Formål
NTNU bruker nå nettsiden Inspera for å utvikle digitale programmerings-eksamener. Ett av oppgavtypene som professorer lager til eksamenene er dra-og-slipp oppgaver (Parsons problems). Disse må i dag lages manuelt og kan være en tidkrevende prosess. I vår forskning tilbyr vi et program som automatisk genererer slike dra-og-slipp oppgaver for Inspera. Forskningen vår går derfor ut på å se om dette programmet gjør prosessen mer effektivt eller ikke. Dette vil gjøres gjennom å observere prosessen mens forsøkspersonene prøver den manuelle og automatiske prosessen, og dermed intervjue de om opplevelsen etter eksperimentet.

Research question:
What is the effect of using this IT artefact to generate Parsons Problems for digital programming exams in Inspera, compared to the manual method?
To what degree does this IT artefact improve the efficiency of generating Parsons Problems for the course supervisor, and how user-friendly is this process?

Denne forskningen er en del av en masteroppgave og all innsamlet data vil kun bli brukt til dette prosjektet.

Hvem er ansvarlig for forskningsprosjektet?
Institutt: NTNU IDI (Institutt for datateknologi og informatikk)
Prosjektansvarlige:

- Joachim Jørgensen
- Simon Kvannli
- Veileder til masteroppgave: Guttorm Sindre

Hvorfor får du spørsmål om å delta?
Utvalgskriteriene for denne forskningen er faglærere og professorer med erfaring, ansvar eller interesse i å utvikle digitale programmerings-oppgaver eller eksamener i Inspera.
Dette utvalget gjøres ved anbefalinger om relevante forsøkspersoner (Snowball sampling technique) originalt fra veileder av masteroppgaven.

Hva innebærer det for deg å delta?
Hvis du velger å delta i prosjektet, innebærer det at vi avtaler et møte hvor vi gir deg noen dra-og-slipp oppgaver som du skal lage i Inspera med og uten vårt program. Vi observerer og gjør et videoopptak av prosessen. Etter du har prøvd programmet vil vi gjennomføre et intervju angående din opplevelse av programmet og andre tilbakemeldinger. Hele prosessen vil ta deg ca. 30 minutter.

Det er frivillig å delta
Det er frivillig å delta i prosjektet. Hvis du velger å delta, kan du når som helst trekke samtykke tilbake uten å oppgi noen grunn. Alle opplysninger om deg vil da bli anonymisert. Det vil ikke ha noen negative konsekvenser for deg hvis du ikke vil delta eller senere velger å trekke deg.

Ditt personvern – hvordan vi oppbevarer og bruker dine opplysninger
Vi vil bare bruke opplysningene om deg til formålene vi har fortalt om i dette skrivet. Vi behandler opplysningene konfidensielt og i samsvar med personvernregelverket.

- Behandlingsansvarlige vil være studentene Simon Kvannli og Joachim Jørgensen, med veileder Guttorm Sindre.
- For å sikre personopplysningene vil alle videoopptak lagres på private og låste maskiner hvor ingen andre enn behandlingsansvarlige har tilgang.

Ingen deltakere vil kunne gjenkjennes i masteroppgavens publikasjon. Ingen private opplysninger annet enn navn, stilling og selve videoopptaket vil brukes i forskningen, men ikke publiseres.

I den publiserte masteroppgaven vil kun utvalgskriteriene være tilgjengelig.

Hva skjer med opplysningene dine når vi avslutter forskningsprosjektet?
Prosjektet skal etter planen avsluttes 1. juni 2019. All innsamlet data som videoopptak og notater vil da slettes når prosjektet er avsluttet.

Dine rettigheter
Så lenge du kan identifiseres i datamaterialet, har du rett til:

- innsyn i hvilke personopplysninger som er registrert om deg,
- å få rettet personopplysninger om deg,
- få slettet personopplysninger om deg,
- få utlevert en kopi av dine personopplysninger (dataportabilitet), og
- å sende klage til personvernombudet eller Datatilsynet om behandlingen av dine personopplysninger.

Hva gir oss rett til å behandle personopplysninger om deg?
Vi behandler opplysninger om deg basert på ditt samtykke.

På oppdrag fra NTNU IDI har NSD – Norsk senter for forskningsdata AS vurdert at behandlingen av personopplysninger i dette prosjektet er i samsvar med personvernregelverket.

Hvor kan jeg finne ut mer?
Hvis du har spørsmål til studien, eller ønsker å benytte deg av dine rettigheter, ta kontakt med:

- NTNU IDI
    - Student: Simon Kvannli (simonkvannli@gmail.com)

- Veileder: Guttorm Sindre (guttorm.sindre@ntnu.no)

- NSD – Norsk senter for forskningsdata AS, på epost (personvernombudet@nsd.no) eller telefon: 55 58 21 17.

Med vennlig hilsen

Guttorm Sindre	Joachim Jørgensen og Simon Kvannli
(Forsker/veileder)	(Masterstudentene)

Samtykkeerklæring

Samtykke kan innhentes skriftlig (herunder elektronisk) eller muntlig. NB! Du må kunne dokumentere at du har gitt informasjon og innhentet samtykke fra de du registrerer opplysninger om. Vi anbefaler skriftlig informasjon og skriftlig samtykke som en hovedregel.

- *Ved skriftlig samtykke på papir, kan du bruke malen her.*
- *Ved skriftlig samtykke som innhentes elektronisk, må du velge en fremgangsmåte som gjør at du kan dokumentere at du har fått samtykke fra rett person (se veiledning på NSDs nettsider).*
- *Hvis konteksten tilsier at du bør gi muntlig informasjon og innhente muntlig samtykke (f.eks. ved forskning i muntlige kulturer eller blant analfabeter), anbefaler vi at du tar lydopptak av informasjon og samtykke.*

Hvis foreldre/verge samtykker på vegne av barn eller andre uten samtykkekompetanse, må du tilpasse formuleringene. Husk at deltakerens navn må fremgå.

Tilpass avkryssingsboksene etter hva som er aktuelt i ditt prosjekt. Det er mulig å bruke punkter i stedet for avkryssingsbokser. Men hvis du skal behandle særskilte kategorier personopplysninger og/eller de fire siste punktene er aktuelle, anbefaler vi avkryssingsbokser pga. krav om eksplisitt samtykke.

Jeg har mottatt og forstått informasjon om prosjektet *(sett inn tittel)*, og har fått anledning til å stille spørsmål. Jeg samtykker til:

- ☐ å delta i *(sett inn aktuell metode, f.eks. intervju)*
- ☐ å delta i *(sett inn flere metoder, f.eks. spørreskjema) – hvis aktuelt*
- ☐ *at lærer kan gi opplysninger om meg til prosjektet – hvis aktuelt*
- ☐ *at mine personopplysninger behandles utenfor EU – hvis aktuelt*
- ☐ *at opplysninger om meg publiseres slik at jeg kan gjenkjennes (beskriv nærmere) – hvis aktuelt*
- ☐ *at mine personopplysninger lagres etter prosjektslutt, til (beskriv formål) – hvis aktuelt*

Jeg samtykker til at mine opplysninger behandles frem til prosjektet er avsluttet, ca. *(oppgi tidspunkt)*

(Signert av prosjektdeltaker, dato)

Tillatelse til å behandle opplysninger på private enheter

Ved å signere på dette dokumentet godkjenner du at all datainnsamling kan oppbevares på private lagringsenheter. Disse private enhetene er passordbeskyttet.
Behandlingsansvarlig institusjon er NTNU IDI (Institutt for datateknologi og informatikk).

Ditt personvern – hvordan vi oppbevarer og bruker dine opplysninger
Vi behandler opplysningene konfidensielt og i samsvar med personvernregelverket.

- Behandlingsansvarlige vil være masterstudentene Simon Kvannli og Joachim Jørgensen, med veileder Guttorm Sindre.
- For å sikre personopplysningene vil alle videopptak lagres på private og låste maskiner hvor ingen andre enn behandlingsansvarlige har tilgang.

Ingen deltakere vil kunne gjenkjennes i masteroppgavens publikasjon. Ingen private opplysninger annet enn navn, stilling og selve videoopptaket vil brukes i forskningen, men ikke publiseres.

I den publiserte masteroppgaven vil kun utvalgskriteriene være tilgjengelig.

Hva skjer med opplysningene dine når vi avslutter forskningsprosjektet?
Prosjektet skal etter planen avsluttes 1. juni 2019. All innsamlet data som videoopptak og notater vil da slettes når prosjektet er avsluttet.

Dine rettigheter
Så lenge du kan identifiseres i datamaterialet, har du rett til:

- innsyn i hvilke personopplysninger som er registrert om deg,
- å få rettet personopplysninger om deg,
- få slettet personopplysninger om deg,
- få utlevert en kopi av dine personopplysninger (dataportabilitet), og
- å sende klage til personvernombudet eller Datatilsynet om behandlingen av dine personopplysninger.

Hva gir oss rett til å behandle personopplysninger om deg?
Vi behandler opplysninger om deg basert på ditt samtykke.

På oppdrag fra NTNU IDI har NSD – Norsk senter for forskningsdata AS vurdert at behandlingen av personopplysninger i dette prosjektet er i samsvar med personvernregelverket.

Hvor kan jeg finne ut mer?
Hvis du har spørsmål til studien, eller ønsker å benytte deg av dine rettigheter, ta kontakt med:

- NTNU IDI
  - Student: Simon Kvannli (simonkvannli@gmail.com)
  - Veileder: Guttorm Sindre (guttorm.sindre@ntnu.no)
- NSD – Norsk senter for forskningsdata AS, på epost (personvernombudet@nsd.no) eller telefon: 55 58 21 17.

Med vennlig hilsen

Guttorm Sindre (Forsker/veileder)	Joachim Jørgensen og Simon Kvannli (Masterstudentene)

Bibliography

Adair, J. G. (1984). The hawthorne effect: a reconsideration of the methodological artifact. *Journal of applied psychology*, 69(2):334.

ANSI (2001). Common industry format for usability test reports.

Atkinson, M. D. (1999). Restricted permutations. *Discrete Mathematics*, 195(1-3):27–38.

Atkinson, R. and Flint, J. (2001). Accessing hidden and hard-to-reach populations: Snowball research strategies. *Social research update*, 33(1):1–4.

Bangor, A., Kortum, P. T., and Miller, J. T. (2008). An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6):574–594.

Beck, K. and Gamma, E. (2000). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Benbunan-Fich, R. (2001). Using protocol analysis to evaluate the usability of a commercial web site. *Information & management*, 39(2):151–163.

Bennedsen, J. and Caspersen, M. E. (2007). Failure rates in introductory programming. *ACM SIGcSE Bulletin*, 39(2):32–36.

Bergin, S. and Reilly, R. (2005). The influence of motivation and comfort-level on learning to program.

Biggs, J. (2014). Constructive alignment in university teaching. *HERDSA Review of higher education*, 1(5):5–22.

Bjork, R. (2017). Creating desirable difficulties to enhance learning. *Best of the Best: Progress*.

Brooke, J. et al. (1996). Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7.

Card, S. K., Moran, T. P., and Newell, A. (1980). The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):396–410.

Clark, R. C. and Mayer, R. E. (2016). *E-learning and the science of instruction: Proven guidelines for consumers and designers of multimedia learning*. John Wiley & Sons.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

Davis, A. M. (1992). Operational prototyping: A new development approach. *IEEE software*, 9(5):70–78.

Denny, P., Luxton-Reilly, A., and Simon, B. (2008). Evaluating a new exam question: Parsons problems. In *Proceedings of the fourth international workshop on computing education research*, pages 113–124. ACM.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, 15(10):859–866.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73.

Ericson, B. J., Foley, J. D., and Rick, J. (2018). Evaluating the efficiency and effectiveness of adaptive parsons problems. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 60–68. ACM.

Ericson, B. J., Margulieux, L. E., and Rick, J. (2017). Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research*, pages 20–29. ACM.

Frøkjær, E., Hertzum, M., and Hornbæk, K. (2000). Measuring usability: are effectiveness, efficiency, and satisfaction really correlated? In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 345–352. ACM.

Garner, S. (2001). A tool to support the use of part-complete solutions in the learning of programming. In *Proceeding de conférence*, pages 222–228. Citeseer.

Garner, S. (2002a). The learning of plans in programming: A program completion approach. In *Computers in Education, 2002. Proceedings. International Conference on*, pages 1053–1057. IEEE.

Garner, S. (2002b). Reducing the cognitive load on novice programmers. In *EdMedia: World Conference on Educational Media and Technology*, pages 578–583. Association for the Advancement of Computing in Education (AACE).

Ihantola, P. and Karavirta, V. (2011). Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education*, 10:119–132.

Ihantola, P. and Karavirta, V. (2019). js-parsons github. [Online]. Available from: https://js-parsons.github.io/. [Accessed 7th March 2019].

IMS Global Learning Consortium. *IMS Question and Test Interoperability (QTI): Overview Version 2.2*. [Online]. Available from: http://www.imsglobal.org/question/qtiv2p2/imsqti_v2p2_oview.html. [Accessed 3rd September 2018].

Inspera AS (2019). Inspera AS home page. [Online]. Available from: http://www.inspera.com/. [Accessed 5th March 2019].

James, G. (1979). The ecological approach to visual perception. *Dallas: Houghtom Mifflin*, pages 127–137.

Jenkins, T. (2002). On the difficulty of learning to program. In *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, volume 4, pages 53–58. Citeseer.

Jenny, B. and Kelso, N. V. (2007). Color design for the color vision impaired. *Cartographic perspectives*, (58):61–67.

Kalvin, A. D. and Varol, Y. L. (1983). On the generation of all topological sortings. *Journal of Algorithms*, 4(2):150–162.

Knuth, D. E. and Szwarcfiter, J. L. (1974). A structured program to generate all topological sorting arrangements. *Information Processing Letters*, 2(6):153–157.

Kurosu, M. and Kashimura, K. (1995). Apparent usability vs. inherent usability: experimental analysis on the determinants of the apparent usability. In *Conference companion on Human factors in computing systems*, pages 292–293. ACM.

Lay, S. (2004). Question and test interoperability: introducing version 2 of the ims qti specification.

Lidwell, W., Holden, K., and Butler, J. (2010). *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub.

March, S. T. and Smith, G. F. (1995). Design and natural science research on information technology. *Decision support systems*, 15(4):251–266.

Nielsen, J. (1994). *Usability engineering*. Elsevier.

Nielsen, J. and Landauer, T. K. (1993). A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 206–213. ACM.

Norman, D. A. (1999). Affordance, conventions, and design. *interactions*, 6(3):38–43.

Oates, B. J. (2005). *Researching information systems and computing*. Sage.

Paas, F., Renkl, A., and Sweller, J. (2003). Cognitive load theory and instructional design: Recent developments. *Educational psychologist*, 38(1):1–4.

Parasuraman, R. and Riley, V. (1997). Humans and automation: Use, misuse, disuse, abuse. *Human factors*, 39(2):230–253.

Parasuraman, R., Sheridan, T. B., and Wickens, C. D. (2000). A model for types and levels of human interaction with automation. *IEEE Transactions on systems, man, and cybernetics-Part A: Systems and Humans*, 30(3):286–297.

Parsons, D. and Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 157–163. Australian Computer Society, Inc.

Piotrowski, M. (2009). Document-oriented e-learning components.

Piotrowski, M. (2011). Qti: A failed e-learning standard? In *Handbook of Research on E-Learning Standards and Interoperability: Frameworks and Issues*, pages 59–82. IGI Global.

Preece, J., Rogers, Y., and Sharp, H. (2015). *Interaction design: beyond human-computer interaction*. John Wiley & Sons.

Rovira, E., McGarry, K., and Parasuraman, R. (2007). Effects of imperfect automation on decision making in a simulated command and control task. *Human Factors*, 49(1):76–87.

Sauro, J. and Kindlund, E. (2005). A method to standardize usability metrics into a single score. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 401–409. ACM.

Sindre, G. and Vegendla, A. (2015a). E-exams and exam process improvement. In *NIK*.

Sindre, G. and Vegendla, A. (2015b). E-exams versus paper exams: A comparative analysis of cheating-related security threats and countermeasures. In *Norwegian Information Security Conference (NISK)*, volume 8, pages 34–45.

Smith, S. L. and Mosier, J. N. (1986). Guidelines for designing user interface software. Technical report, Citeseer.

Sommerville, I. (2011). *Software engineering 9th Edition*. Addison-Wesley Publishing Company.

Sugimori, Y., Kusunoki, K., Cho, F., and Uchikawa, S. (1977). Toyota production system and kanban system materialization of just-in-time and respect-for-human system. *The international journal of production research*, 15(6):553–564.

Sweller, J., Van Merrienboer, J. J., and Paas, F. G. (1998). Cognitive architecture and instructional design. *Educational psychology review*, 10(3):251–296.

Tavangarian, D., Leypold, M. E., Nölting, K., Röser, M., and Voigt, D. (2004). Is e-learning the solution for individual learning?. *Electronic Journal of E-learning*, 2(2):273–280.

Tilkov, S. and Vinoski, S. (2010). Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83.

Todd, P. A. and Benbasat, I. (1987). Process tracing methods in decision support systems research: Exploring the black box. *Mis Quarterly*, 11(4):493–512.

Van Merriënboer, J. J. (1990). Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of educational computing research*, 6(3):265–285.

Watson, C. and Li, F. W. (2014). Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 39–44. ACM.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83.

Williams, L., Kessler, R. R., Cunningham, W., and Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE software*, 17(4):19–25.

Jørgensen & Kvannli

Efficient Generation of Parsons Problems for Digital Programming Exams in Inspera