CHALMERS
UNIVERSITY OF TECHNOLOGY

BACHELOR'S THESIS IN COMPUTER SCIENCE

Constructing a Game Modelled After Logic and Proofs

TENJIN: A SMARTPHONE GAME

Elias Burström, Jonatan Källman, Tuyen Ngo & Felix Nordén

Department of Computer Science
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Bachelor's thesis 2019

Tenjin

A Smartphone Game for Logic and Proofs

Elias Burström, Jonatan Källman, Tuyen Ngo & Felix Nordén

Department of Computer Science
Chalmers University of Technology
Gothenburg, Sweden 2019

Tenjin
A Smartphone Game for Logic and Proofs
Elias Burström, Jonatan Källman, Tuyen Ngo & Felix Nordén

Supervisor: Andreas Abel, Department of Computer Science and Engineering
Examiners:
Wolfgang Ahrendt, Department of Computer Science and Engineering
Morten Fjeld, Department of Computer Science and Engineering
Sven Knutsson, Department of Computer Science and Engineering

Bachelors's Thesis 2019
Department of Computer Science
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Concept art of the smartphone game *Tenjin.*

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2019

Abstract

Logical reasoning plays an important role in human behaviour and is widely used in everyday life. It is incorporated in contexts such as natural languages, mathematics and programming. However, the general knowledge in today's society with regards to formal logic is lacking and does not reflect the important role it plays. This lack of knowledge may not be an issue in simple situations. However, in more complex situations, not having a formal understanding may lead to erroneous reasoning. One explanation for this knowledge gap may be the lack of formal education of logic in school, with the bulk being taught as late as in university. Tenjin, a smartphone game modelled after classical propositional logic in the style of sequent calculus, tries to mitigate this problem. Tenjin aims to make learning formal logic accessible to people in the age range of 13–25 by replacing the formal syntactic notation with an intuitive space-themed user interface. As a proof of concept for Tenjin, this paper describes a development- and design process which results in both an accurate software representation for a subset of classical propositional logic and its proof rules, and a tested and intuitive visualisation for users to interact with. Designing the visualisation was done through incremental testing. First, each design-candidate went through benchmark-tests using proofs of different difficulty. If a candidate passed the benchmark-tests, it then went through paper prototype-tests with real-life users. Development of the software was executed using well-founded software principles and tools, e.g. the S.O.L.I.D principles and different design patterns. After performing a thorough modelling process of the problem domain, the code-base was written in the programming language C# and the frameworks .NET and Unity.

Keywords: user experience, graphic design, software engineering, software development, classical propositional logic, sequent calculus.

Sammandrag

Logiskt resonemang spelar en viktig roll för mänskligt beteende och används i många aspekter av vardagslivet. Det är en del av ämnen såsom naturliga språk, matematik och programmering. Allmänhetens knappa kunskap om ämnet speglar däremot inte dess betydelse. I enklare situationer behöver denna brist på kunskap inte vara ett problem, men i mer komplexa situationer kan en avsaknad av kunskap leda till felaktiga resonemang. En förklaring till denna kunskapsbrist kan vara avsaknaden av logik i den allmänna skolan, då ämnet huvudsakligen lärs ut på universitetsnivå. Detta problem skulle kunna mildras med hjälp av Tenjin, ett smartphone-spel baserat på klassisk satslogik i formen av sekventiell analys. Tenjin ämnar göra inlärningsprocessen av formell logik tillgänglig för individer i åldrarna 13–25 genom att ersätta den formella syntaktiska notationen med ett intuitivt, rymd-inspirerat användargränssnitt. För att agera på ett konceptuellt plan för Tenjin, så beskriver denna tes en utvecklings- och designprocess som resulterar i en precis mjukvarurepresentation för en delmängd av klassisk satslogik, samt en testad och intuitiv visualisering för användaren att interagera med. Design av den visuella representationen genomfördes med hjälp utav inkrementell testning. Först testades varje kandidat genom prestanda-tester som baserades på bevis av olika svårighetsgrad. Om en kandidat klarade prestanda-testerna, så testades den sedan genom pappers-prototyptester med riktiga användare. Utveckling av mjukvaran utfördes med hjälp av välgrundade utvecklingsprinciper och verktyg, såsom S.O.L.I.D-principerna och diverse designmönster. Efter en genomgående modelleringsprocess av problemdomänen användes programmeringsspråket C# samt ramverken .NET och Unity för att skriva kodbasen.

Nyckelord: user experience, grafisk design, informationsteknik, mjukvaruutveckling, klassisk satslogik, sekventiell analys.

Acknowledgements

First and foremost, we would like to thank our supervisor Andreas Abel for proposing this project, as well as giving us insightful feedback and input during its execution. We would also like to give acknowledgement to the people from the department for technical language and Chalmers Writing Centre, who gave us useful suggestions and guidance throughout the thesis.

Contents

List of Definitions

Syntactic notation Is used when talking about the structure of how a language is represented, i.e. its grammar. In this report we focus on the structure of formal logic, e.g. the symbols and lexical ordering.

Semantics Is used when talking about the underlying meaning of a sentence or statement. Focus will be of the semantics of logic statements, e.g. $A \vee B$ is the *syntactic* notation of "A or B", but the *semantic* value is "For this statement to hold, at least one of A or B is true".

Problem Domain Refers to the project domain under the defined constraints, i.e. logic and proofs within classical propositional logic.

Domain In general terms, it refers to a specific area given a context. Except for in the introduction/background, this term is almost certainly always synonymous to the term *Problem domain*.

Polymorphism Refers to the concept of an object having multiple shapes. In programming this is used to describe an object being interpreted to have different types, e.g. subtypes or interface types.

Composition Refers to the concept of letting an object have a reference to another object.

Domain specific language Refers to a specialised language developed to facilitate the communication in a particular domain.

Big ball of mud Synonymous with having a code base which is so complex that it is no longer manageable.

Aliasing Having multiple symbolic references bound to the same structure. Modifying the structure from one such reference will result in the change spreading to all other references, which can yield unsolicited behaviour.

Bus factor The number of people in a project required to be run over by a bus for the project to discontinue.

Left/right disjunction Left/right disjunction refers to a disjunction that is on the left-

/right side of the turnstile.

Left/right conjunction Left/Right conjunction refers to a conjunction that is on the left/right side of the turnstile.

List of Figures

List of Tables

1

Introduction

Logical reasoning is a fundamental building block of human intelligence and has played a big part in human progress. Today, logic has been incorporated in many different contexts, e.g. natural languages, argumentation, mathematics, and, as a result of digitisation, computers and programming. Even though logic has become widespread and integrated into many areas of everyday life, not many know the formal structure of logic and instead reason on an intuitive level. Intuitive reasoning may prove efficient when facing simpler problems, but may become insufficient and erroneous when faced with problems of higher complexity. This may become problematic, as there are situations where forming the wrong conclusion given a set of premises can become very costly. A classic example of this is the argument from ignorance-fallacy, or *argumentum ad ignorantiam*, where one assumes that if a statement p is unproven, then the statement *not* p must be true [1].

One of the reasons for the lacking knowledge in formal logical reasoning could be the lack of education of the subject in school, while the bulk of the education takes place at university. Once there, the education tends to be quite dense and abstract, making the learning curve quite steep; this may be another reason for the problem.

In order to both introduce the concept of formal logical reasoning earlier on, and to flatten the learning curve, this project is meant to result in a smartphone game. This game should be capable of introducing formal logical reasoning and proof theory to individuals in their teens or above, in an intuitive fashion.

1.1 Background

In Sweden, the National Agency for Education is responsible for specifying what the Swedish students are to learn up to a certain academic level [2], [3]. The curricula in mathematics show that students in compulsory school, age 7–15, do not come in contact with logical reasoning and proofs at all, despite its importance [4]. For the students in upper secondary school, age 16–19, there are some mentions in the curriculum regarding the terminology used in logic, and a few geometry-related proofs that are to be taught [5].

In later years, the National Agency for Education has put more emphasis on embedding programming into the curriculum due to the digitisation of society [6]. The increased emphasis has resulted in programming being weaved into the curriculum of mathematics in compulsory school. This can be considered ironic since programming is based on logical reasoning, which the curriculum is lacking. Furthermore, coupling the generality of logic and proofs with the specificity of a certain domain may pose a problem. It may impede the student in seeing how logic itself can be a science.

On another note, due to the wide array of areas in mathematics that need to be covered in the curriculum, not much room remains for learning about logic and proofs at a higher level. This lack of experience can present a problem as the student may be unprepared for their university studies in regards to logic and proofs.

It is curious that logic and proofs are only presented in higher academic settings, despite that its fundamentals can be simplified considerably. Take addition and multiplication, for example, both of these operations can be studied much more in depth at university e.g in group theory, but they are still being formally taught to children at a low level. The difficulty and complexity of these operations increase continuously as the student finishes each year of school. In comparison, logic and proofs are being taught discretely, making its first appearance in upper secondary school, and then takes another big step to the level taught at university. This can make the subject confusing, and hard to learn due to the lack of familiarity.

1.2 Purpose

The purpose of this research is to construct a smartphone game modelled after logic and proofs. This report describes the process of defining the problem domain (def. Problem Domain), modelling the domain into implementable software as well as the implementation of that software. Furthermore, this report describes the process of designing an intuitive visualisation that is representative of the proving process found in classical propositional logic.

1.3 Problem

With the purpose in mind, constructing a smartphone game modelled after logic and proofs can be divided into two separate problems or tasks.

The first task is to accurately implement the domain of propositional calculus as a software model, using good code practices and adhering to the core principles of software engineering.

The second task is to design and create a visualisation that presents the concepts of logic and proofs on an introductory level. In order to reach such a level, the formality and syntactic notation (def. Syntactic notation) used at university need to be removed. Furthermore, the semantics (def. Semantics) should be visualised in an intuitive manner to encourage more interaction. By playing the game, the player comes in contact with the concepts without being confronted by the intimidating notation of formal logic and proofs.

1.3.1 Criteria for a Successful Application

In order to concretise the problem and to measure the success of the application, a set of criteria should be defined. By the time the project is finished, the team strives to have:

1. A fully functional and platform-agnostic backend modelled after propositional logic.

2. The ability to generate proofs in the backend.

3. A complete front-end implementation of the chosen visualisation.

4. At least one more alternative visualisation of the domain.

1.4 Scope and Limitations

Logic and proofs can be very extensive. Logic is categorised in different orders, starting from the zeroth. Each order is built upon the previous one, so the zeroth order logic, also commonly known as propositional logic, will have to be modelled and implemented in the game. However, first-order logics and above will not be considered in this project. More information about propositional logic can be found in section 2.1. Along with different orders of logic, there also exist many different types, e.g. classical logic, constructive logic, and temporal logic, but this project only focuses on classical logic. The intention is to build a solid foundation which can be further extended for other usages in academia, e.g. research in the field of game-based learning or using the smartphone game in educational contexts.

Even though the game could be played on bigger devices such as tablets, the project will target smaller screens when considering the design choices for the visualisation. Since smartphones are more widespread than tablets [7], choosing the smaller screen as a target means a greater potential reach.

Visualisation is a key aspect of the project, as the user interacts with the game through this medium. While there is a multitude of visualisations to represent the semantics of logic, this project has been limited to only consider three visualisations.

As for the general aspects of a smartphone game, the game should be single-player and is to be stored locally on the mobile device. The game targets students between 13 and 25 years old, but it should be accessible to anyone that is interested.

1.5 Social and Ethical Aspects

In regard to social and ethical aspects, a smartphone game can have both positive and negative impact. One negative aspect of a smartphone game could be a potentially higher risk of smartphone addiction. In the case of Tenjin, there have not been any conscious efforts to increase the playtime of a user, e.g. through manipulative algorithms, since there is no interest in it. Furthermore, it is hard to fathom how the initial release of the game could be anywhere close to addictive. On the contrary, Tenjin could potentially add to society as it aims to mitigate the lack of knowledge of an important subject. Additionally, the game could be used as a tool by researchers to study the effects from learning logic and proofs in a visual and gamified context.

1.6 Paper Outline

The remaining structure of this paper is as follows:

First, in chapter 2, *Theory*, the underlying theory of the project is introduced and explained in order to give the necessary context and to create a better understanding for the rest of the paper.

Next, the chapters 3, 4, and 5 which are named *Preliminary Process and Project Preparation*, *Creating the Different Visualisations*, and *Developing the Application* respectively, present and justify the methods and procedures used in each of the three project phases.

Then, in chapter 6, *Visualisations of Sequent Calculus*, the results from the Visualisation Phase are presented. Similarly, the following chapter, *Tenjin Core: The Brains of the Operation*, describes the developed logic module of the application in a conceptual manner. The succeeding chapter after that, *Tenjin as the Resulting Application*, showcases the end-result of the application created by merging the final visualisation and the logic module.

After the results, the chapter *Discussion* analyses and deliberates upon the results and the choice of methods which led up to them.

Finally, chapter 10, *Conclusions*, presents the conclusions drawn from the process of constructing a smartphone game based on logic and proofs.

2

Theory

The logic and the proof calculus which the game is based on will be explained thoroughly in the following sections. This chapter will also describe the necessary information regarding the underlying techniques required to implement the game.

2.1 Propositional Logic

To get an understanding of what propositional logic entails, it is necessary to first know what a proposition is. The classical answer is: a proposition is a statement that has a truth value of either true or false. More concretely, a proposition can be conveyed in the form of a descriptive sentence, such as the ones below:

1. Water is wet.

2. The sun is not shining.

3. It is blue and green.

Do note that this report uses the words *statement* and *proposition* interchangeably. A proposition consists of one or more propositions, which it can be broken down into. When a proposition no longer can be broken down into other propositions, it is called an *atomic proposition.* Proposition 1 above is an example of such a proposition.

In the case where a proposition consists of multiple other propositions, it is called a *molecular proposition.* The internal propositions of such a proposition are represented through the use of *connectives.* Connectives, or *logical operators*, are operators of different arity with different semantic values.

Out of the different connectives available, the main connectives used in propositional logic [8] are listed in table 2.1. Proposition $A \leftrightarrow B \rightarrow C \vee \neg D \wedge E$ is thus to be read as $A \leftrightarrow (B \rightarrow (C \vee ((\neg D) \wedge E)))$.

Table 2.1: The five logical operators in propositional logic [9].

Operator	**Natural language**	**Name**
$\neg$	not	Negation
$\wedge$	and	Conjunction
$\vee$	or	Disjunction
$\rightarrow$	if ... then ...	Implication
$\leftrightarrow$	if and only if	Biconditional

With the help of table 2.1, proposition 2 can be seen as a molecular proposition, as it contains the connective *not*. Proposition 3 can also be seen as a molecular proposition, as it consists of the two propositions "It is blue" and "It is green", with the logical connective *and* in between.

Yet, it could also be argued that these propositions are atomic propositions respectively, if only their truth value is of interest. Throughout the report, if a proposition can be broken down into other propositions, it will be represented as a molecular proposition, regardless of there being an interest in the constituting atomic propositions or not.

By applying the aforementioned operators to propositions, more complex *logical formulas* are formed. For example, proposition 3 in the list of examples above could be written as $A \wedge B$, where A represents "It is blue" and B represents "It is green". In general, atomic propositions are represented by capital letters, e.g. A, B or C, for the sake of convenience and brevity [8]. Furthermore, when reading logical formulas in natural language, one replaces the connective with its semantic value. For example, $A \wedge B$ is read "A *and* B" while $\neg B \vee C$ translates to "*not* B *or* C' [9].

Propositional logic comes in several flavours. The most common form is classical logic, but there non-classical logics, the most prominent being intuitionistic logic. For this project, classical logic was chosen, cf. section 3.2. In classical logic, implication $A \rightarrow B$ can be defined as $\neg A \vee B$. Similarly, conjunction and disjunction are interdefinable via the de Morgan laws, but this shall not matter for the sake of this project. Uniformly in classical and intuitionistic logic, the biconditional $A \leftrightarrow B$ is definable as $(A \rightarrow B) \wedge (B \rightarrow A)$.

The choice of logic, e.g., classical vs. intuitionistic, is especially important when considering the respective proof calculi, which can be substantially different. In the next section, the proof calculus is introduced that is relevant for the project: classical sequent calculus à la Gentzen [10].

2.2 Sequent Calculus

A *sequent* in sequent calculus can be explained as a pair of two sequences of logical formulas, separated by a turnstile. The sequence to the left of the turnstile is commonly called antecedent, and its formulas hypotheses or premises. The sequence to the right is conversely called consequent, and its formulas are called conclusions [11].

Essentially, a sequent $\gamma_1, ..., \gamma_n \vdash \phi_1, ..., \phi_m$ is a conjunction of premises γ_i which semantically entails a disjunction of the conclusions ϕ_j, where $n, m \in \mathbb{N}$ and $i = 1..n$ and $j = 1..m$. Commonly, sequences are denoted by upper case Greek letters like Γ and Δ, allowing sequents to be written as $\Gamma \vdash \Delta$. One should note that the premises or conclusions of a sequent may be empty [12], which is then written as $\Gamma \vdash$ or $\vdash \Delta$. Here, both Γ and Δ represent the "environments" of the premises and conclusions respectively, which may or may not include any other propositions that are irrelevant for the proof. One can add propositions to Γ or Δ while preserving the truth value of the sequent, making the proof possibly easier to solve. However, this also makes the proof less general, and is referred to as "weakening" [13].

Out of the rules and operators available in sequent calculus, this project uses the elimination rules of the logical operators *conjunction*, *disjunction*, and *negation*. The elimination rules for the chosen operators are defined differently depending on the logical operator being part of the premises or the conclusions, i.e. whether they are to the left or to the right of the turnstile. These rules are formally stated in the table below.

Table 2.2: The left and right elimination rules of each logical operand in sequent calculus [14].

	(Left)	*(Right)*
Negation Rule	$\dfrac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}(\neg l)$	$\dfrac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}(\neg r)$
And Rule	$\dfrac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}(\wedge l)$	$\dfrac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}(\wedge r)$
Or Rule	$\dfrac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}(\vee l)$	$\dfrac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}(\vee r)$

A sequent is axiomatically true, i.e. true without needing a proof, if two identical atomic propositions exist on either side of turnstile. Notationally, an axiomatically true sequent is shown by adding a bar above it. For example, $\overline{\Gamma, A \vdash A, \Delta}$ is axiomatically true, as there is one instance of A on either side of the turnstile. In contrast, the sequent $\Gamma, A, (\neg A \vee B) \vdash B, \Delta$ is not axiomatically true, unless $B \in \Gamma$ or $A \in \Delta$. However, by applying the elimination rules in table 2.2, the sequent can be proven since all of the sub-proofs generated from the elimination rules are proven. The next two sections showcase the process of proving two different sequents.

It is worth noting that when proving a sequent through elimination, the resulting sequents from applying the eliminating rules are shown above the bar. The workflow is bottom up, as can be seen in table 2.2, and the two respective examples in sections 2.2.1 and 2.2.2. Additionally, the used elimination rule is shown to the right of the bar of a sequent. With this workflow, the structure resembles a tree, so the whole proof's structure is called a proof tree.

Table 2.2 also shows that $\vee l$ and $\wedge r$ generate two sequents above the bar. This means that in order to eliminate a disjunction on the left side of the turnstile, or conjunction on the right side of the turnstile, the two propositions which make up the molecular proposition also need to proven. On the other hand, eliminating $\wedge$ on the left side of the turnstile, or $\vee$ on the right side, is more trivial. This is because premises are “conjunctions of propositions”, and conclusions are “disjunctions of propositions”. $\Gamma, A \wedge B \vdash \Delta$ is equivalent to $\Gamma, A, B \vdash \Delta$, similar to how $\Gamma \vdash A \vee B, \Delta$ is equivalent to $\Gamma \vdash A, B, \Delta$.

2.2.1 Example I: Proving a Simple Sequent

$$\Gamma, A, (\neg A \vee B) \vdash B, \Delta$$

As written in the previous section, this sequent is not axiomatically true—except when $B \in \Gamma$ or $A \in \Delta$. However, it should be derivable, as in light of the equivalence of $\neg A \vee B$ and $A \to B$ it states the well-known principle of *modus ponens*: If A and $A \to B$ both hold, then B holds as well.

To prove this sequent, the left elimination rule for disjunction can be used, leading to the following partial proof tree:

$$\frac{\Gamma, A, \neg A \vdash B, \Delta \qquad \overline{\Gamma, A, B \vdash B, \Delta}}{\Gamma, A, (\neg A \vee B) \vdash B, \Delta}\,(\vee l)$$

It can be seen that $\overline{\Gamma, A, B \vdash B, \Delta}$ is axiomatically true, and therefore has a bar over it. After all, it has the atomic proposition B on both sides of the turnstile. The proof is not finished, however. For the proof to be finished, all of the sequents need to have a bar above themselves, so $\Gamma, A, \neg A \vdash B, \Delta$ needs to be proven as well.

$$\frac{\dfrac{\overline{\Gamma, A \vdash A, B, \Delta}}{\Gamma, A, \neg A \vdash B, \Delta}\,(\neg l) \qquad \overline{\Gamma, A, B \vdash B, \Delta}}{\Gamma, A, (\neg A \vee B) \vdash B, \Delta}\,(\vee l)$$

Applying the left elimination rule for negation leads to the sequent $\overline{\Gamma, A \vdash A, B, \Delta}$, which is axiomatically true as A appears on each side of the turnstile. Since all of the sequents have been proven, the sequent $\Gamma, A, (\neg A \vee B) \vdash B, \Delta$ is therefore proven.

2.2.2 Example II: Solving an Intermediate Proof

$$\Gamma, (A \vee B) \wedge C, \neg B \vee C \vdash B \vee C, \Delta$$

It is clear that this sequent is not axiomatically true. There are multiple ways to start working on this sequent, but it is a good idea to use either the rules $\wedge l$ or $\vee r$ if possible, as they only generate one new sequent, opposed to two when using the rules $\vee r$ or $\wedge l$. Starting with the $\wedge l$-rule for $(A \vee B) \wedge C$, the following partial proof tree is generated:

$$\dfrac{\Gamma, A \vee B, C, \neg B \vee C \vdash B \vee C, \Delta}{\Gamma, (A \vee B) \wedge C, \neg B \vee C \vdash B \vee C, \Delta}\ (\wedge l)$$

C is now an atomic proposition of the premises. It is appropriate to then use $\vee r$, so that C also can be an atomic proposition conclusions:

$$\dfrac{\dfrac{\overline{\Gamma, A \vee B, C, \neg B \vee C \vdash B, C, \Delta}}{\Gamma, A \vee B, C, \neg B \vee C \vdash B \vee C, \Delta}\ (\vee r)}{\Gamma, (A \vee B) \wedge C, \neg B \vee C \vdash B \vee C, \Delta}\ (\wedge l)$$

By doing so, there are two identical atomic propositions on either side of the turnstile, and the sequent has thus been proven.

2.3 The Programming Language C#

Filling the role of a type-safe and object-oriented language, C# allows developers to construct programs that make use of the .NET Framework. The language can be used for constructing many different applications, program components and web services. It is similar to languages like C, C++ and Java, with slight differences in certain features. The core of the application, also referred to as the logic module, is written in C#.

2.3.1 C# Programming in .NET

Within the .NET framework there is an assortment of different project types [15, Sec. Arguments], all of which get compiled and distributed as *assemblies* (either `.exe` or `.dll` files) that other .NET projects may depend on. One such project type is class libraries, which consist of different modules that are imported as their functionality is needed. Provided by these modules are functions such as reading and writing files, graphic related functions, database handling and much more. It is a toolkit which provides the program with predefined solutions to problems.

To manage the execution of programs that make use of the .NET implementations, the interpreter Common Language Runtime (CLR) is used. The CLR is considered a part of the .NET framework [16]. Unlike projects written in Java, projects that are written in C# do not have a packaging structure that depends on the file structure. Instead, all modules are defined within namespaces, which act as virtual packages in order to add a scoping layer [17]; these namespaces can be referenced by other modules. This means that modules that exist in different sections of the file system still can exist in the same namespace, which is a powerful structuring principle that allows extensiblity of already exisiting code bases. Furthermore, not having a directory-based package structure means that internal packaging is not possible and therefore there is no Java-like “package private” access modifier in C#. Instead, the default access modifier for classes and interfaces are assembly-wide, meaning that the assembly itself will become the only package-like structure of a .NET project.

2.3.2 Different Implementations of .NET

Two different .NET implementations are used in this project: .NET Core and .NET Standard. In the program implementation, these parts of the .NET landscape are used in different parts of code. The logic core code of the program uses the .NET Core implementation, but the Unity project only naturally supports the .NET Standard libraries. Thus .NET Standard is used for the scripting done with Unity.

While .NET Core is optimised for mobile and server assignments, it also provides a powerful high-class command-line interface (CLI). Being an open-source and multi-platform implementation of .NET, it gives access to libraries that ease development of cloud services, console apps and web applications.

Every implementation of .NET makes use of .NET Standard in some way. It takes the role of a base specification for what application programming interfaces (APIs) all other .NET stacks should extend. .NET Standard consists of libraries, accessed directly or by other implementations of .NET. As these libraries are accessible from all .NET implementations, .NET Standard provides a consistent functionality across them all [18].

2.4 Unity Game Engine

Unity 3D is the choice of game engine software for this project. It is capable of porting the game to 25 different platforms and is therefore not dependent on any specific platforms. This game is focused on being available on iPhones and Android phones, which is naturally supported by Unity. Unity 3D is a well established and popular game engine that is used for both 2D and 3D game development. [19].

Unity game engine offers a lot of tools for game developers. On their homepage, Unity Technologies themselves describes what a game engine offers with the following description of its features. “...scenes and environments; add lighting, audio, special effects, physics and animation, interactivity, and gameplay logic; and edit, debug and optimise the content for your target platforms. [20]” Each of the general features considered relevant in Unity game engine will be described hereafter in their own subsections.

3
Preliminary Process and Project Preparation

With the underlying theory being presented, this chapter is the start of a block which describes and motivates the methods and the procedures used in the project. It first presents an overview of all of the events leading up to the final product, with the help of the figure 3.1. Then, it describes the first step of the project, which was a literature study on logic and proofs.

3.1 Overview of the Project

Constructing a smartphone application from the ground required an appropriate visualisation of the domain and a functional backend for the application to use. Thus, the procedure of the project occurred in two parallel tracks and was then merged into one in the end to create the final application. Figure 3.1 shows the entire process from start to finish. The different colour blocks separate the content into individual chapters, starting with this chapter.

3.2 Researching Logic and Proofs

As an initial step, a literature study about logic and proofs was made to gain as much information as possible about the domain. Once enough information had been collected, the group had to decide on which type of logic to model the game upon. As presented in section 1.2, classical propositional logic was chosen, as classical logic is the logic which the team was most familiar with and all higher-order logic is based on propositional logic. Furthermore, the style to express classical propositional logic was chosen to be sequent calculus. These decisions are important as they heavily influenced both the visualisation- and implementation aspect of the project. Propositional logic is further explained in section 2.1 and sequent calculus is presented in 2.2.

After choosing the type and order of logic to work with, a set of problems were created to use as a benchmark to test potential visualisations on. These problems' primary purpose was to highlight inconsistencies when representing a sequent and can be found in section B.4 of appendix B.

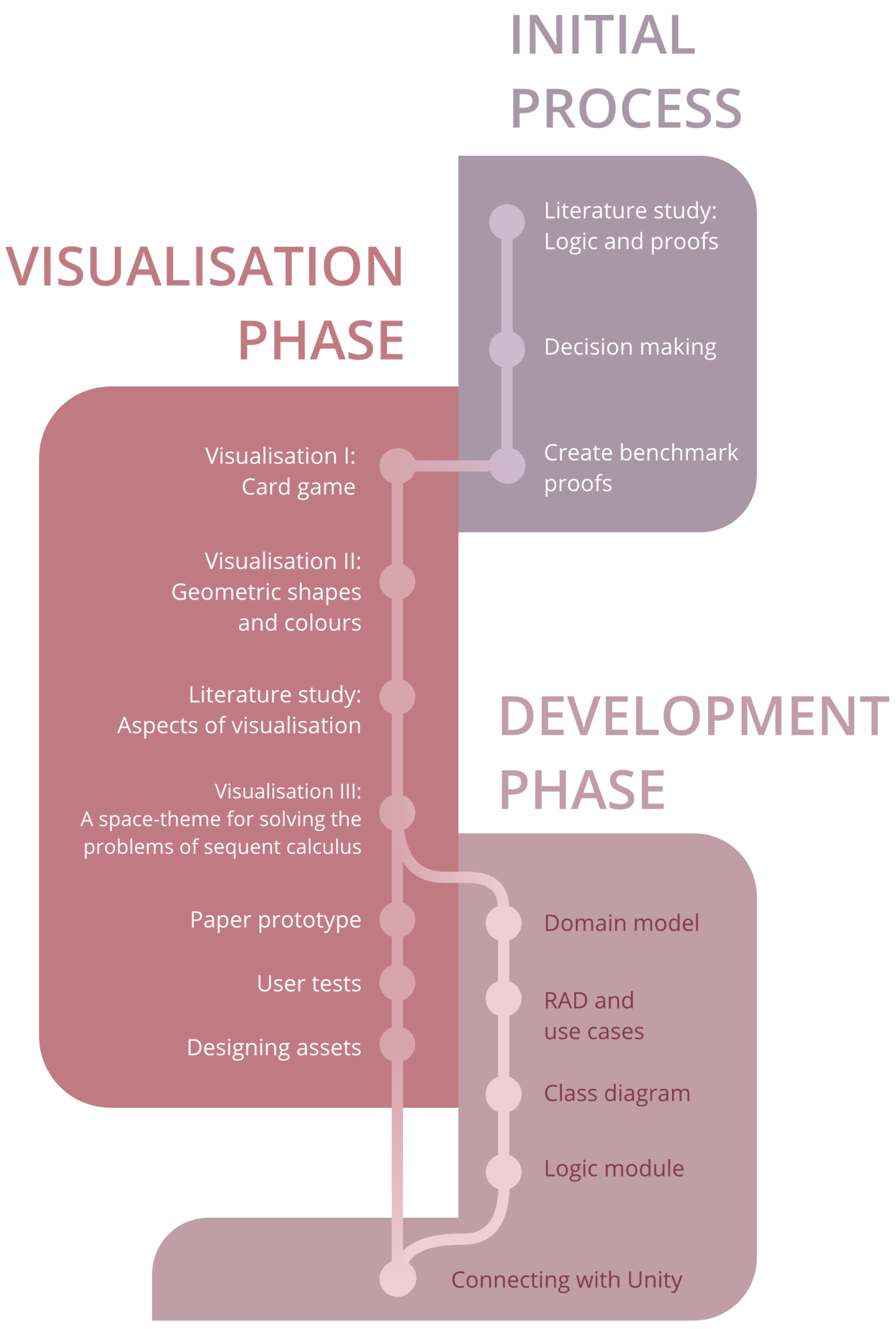

INITIAL PROCESS
Literature study: Logic and proofs
Decision making
Create benchmark proofs
VISUALISATION PHASE
Visualisation I: Card game
Visualisation II: Geometric shapes and colours
Literature study: Aspects of visualisation
Visualisation III: A space-theme for solving the problems of sequent calculus
Paper prototype
User tests
Designing assets
DEVELOPMENT PHASE
Domain model
RAD and use cases
Class diagram
Logic module
Connecting with Unity

Figure 3.1: A diagram showing the process from the start (top of the diagram) to the end of the project (bottom of the diagram).

4 Creating the Different Visualisations

After the initial process of researching the domain, the process of creating visualisations began. This chapter describes the methods and procedures that were used when creating a visual representation for the domain of logic and proofs. The procedure is described chronologically, and show the difficulties that were encountered along the process.

First, the initial visualisation as a card game is presented. Next, it presents the first alternative visualisation, which is based on geometric shapes and colours. Then, a literature study regarding what makes a visualisation intuitive is described. Lastly, the third and final visualisation inspired by space is presented. Along with the creation of the space-themed visualisation, the process of creating a paper prototype and using it when performing user tests is shown.

4.1 Visualisation I: Card Game

When the project started, there was a suggestion to visually represent logic and proofs using a card game. The idea was entertained by the group but was discarded after trying to visualise the benchmark proofs (found in B.4 of appendix B). It had flaws which limited the visual representation.

First of all, it was suggested to represent a disjunction ($A \vee B$) with a rotating card, displaying different ranks on the two sides. This limits disjunctions to only consist of two atomic propositions. For example, $(A \vee B) \vee C$ cannot be visually represented.

Second of all, the size of a smartphone screen may be too small to display many intricate cards. The visualisation could, therefore, become overly complicated and cramped. Thus, the process of creating another visualisation began.

4.2 Visualisation II: Geometric Shapes and Colours

Using the insights gained from the card game visualisation, the second visualisation tried to make use of simple geometric shapes and colours. This visualisation managed to represent the proposition $(A \vee B) \vee C$ and did not take up as much space as the previous one. However, the benchmark problems indicated that the complexity of the visual representation rapidly outgrew the complexity of the problem at hand. Therefore, this visualisation was also discarded, as the visual complexity could entail cognitive overload for the user.

4.3 Conceptualising the Problems: Defining Key Aspects of Intuitive Visualisations

With the two previous visualisations having issues, the consensus was that doing a literature study only on the domain was insufficient. Therefore, the decision was made to take a step back and conduct a visualisation-focused literature study.

Up until this point, the fact that the visualisation should be intuitive had not been taken into consideration. Thus, the literature study was focused on the term "intuitive visualisation" by defining different key aspects associated with the term. From the literature study, the group decided to focus on the aspect of *naïve physics* for the next visualisation. The idea of naïve physics is that it is representative of the world and therefore recognised by everyone, no matter the cultural background [21].

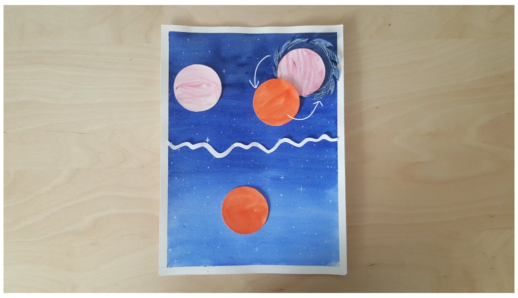
4.4 Visualisation III: A Space-Theme for Solving the Problems of Visualising Sequent Calculus

With naïve physics in mind, figuring out how to incorporate it into the visualisation was the next issue. After discussion, the idea of naïve physics evolved into a focus on physics and chemistry. Inspired by the fact that a proposition can be either *atomic* or *molecular*, as written in section 2.1, the atomic propositions were represented as atoms and molecular propositions as molecules. Then, this idea manifested itself as a space-theme, which was reasoned to be valid since everyone should recognise space.

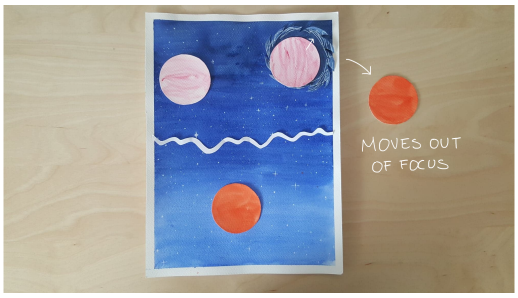
The space-idea was tested with the benchmark problems and succeeded in visually representing the sequents without being too complex. Afterwards, decisions were made regarding the details of the visualisation. As this visualisation was deemed satisfactory, the team then started to branch out to work in parallel. One part of the team began working on the logic module of the game, which can be read more in chapter 5, and the other part of the group continued with the visualisation. What this continuation meant was to create a paper prototype and then test it on users, which can be further read below.

4.4.1 Creating the Paper Prototype

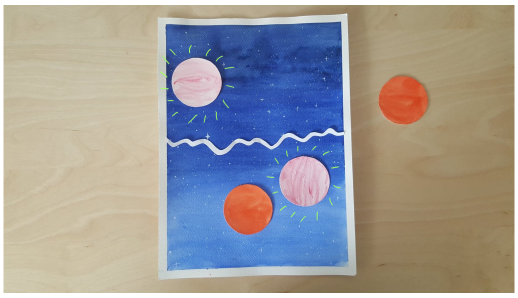
A paper prototype with the space-theme was created to conduct user tests. In short, atomic propositions are planets and negations are "black holes" behind its operand. Additionally, left-disjunctions and right-conjunctions (def. Left/right disjunction and Left/right conjunction respectively) are "force fields" encapsulating its operands, and the operands



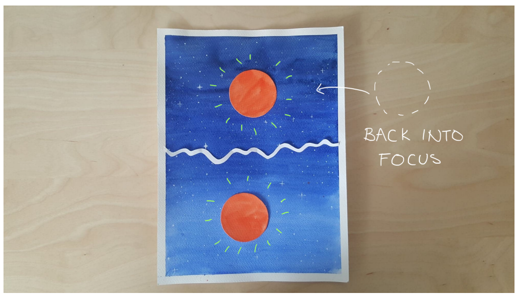
(a) The pink planet with its black hole rotating in an orbit with the orange planet to represent disjunction on the left side of the turnstile.



(b) The rotation is stopped, i.e. the left elimination rule of $\vee$ is applied. The pink planet is focused on, and subsequently moved into the black hole, to reappear on the other side of the rift.



(c) The pink planets are on each side of the rift, both is tapped on, and the user has thus proven the proposition A.



(d) The orange planet is being refocused on, and the user can prove thus prove the proposition B, by tapping on both of the orange planets. After this, the level is finished.

The clause $A, \neg A \vee B \vdash B$ represented as different bodies in space in the game. The pink planet represents A, the black hole behind the other pink planet is the negation $\neg A$, the orange planet represents B, and the rift in the middle of the screen represents the turnstile $\vdash$. Above, the figures show how to solve the simple clause.

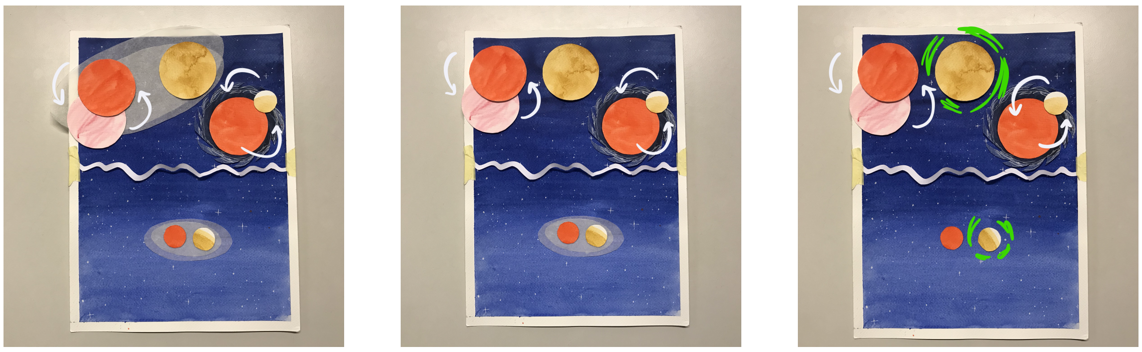
Figure 4.1: The paper prototype modelled after a simple proof.

of both right-disjunctions and left-conjunctions are “orbiting” (or rotating) around the centre point between them. Examples of the paper prototype can be found in figures 4.1 and 4.2.

4.4.2 User Testing of Visualisation III

Before reaching out to the volunteering users, a script was created to ensure that the tests remained consistent for all users. This script can be found in section B.1. Each user was invited to their separate session, since the choice of one user should not influence the choice of another.

All tests were started by explaining the context of the game: A rift has been created in space, and the player’s goal is to repair the rift by matching the available planets on each side of the rift. However, planets can only be matched if they are “free”, i.e. they are not within a force field, rotating in an orbit, nor do they have a black hole behind them.



(a) The pink planet represents A, the orange planet represents B, and the yellow planet represents C.

(b) The bigger force-field is removed, i.e. $(\wedge l)$ is applied.

(c) The force-field on the other side is also removed, i.e. $(\vee r)$ is applied. Both of the yellow planets are free-floating planet on each side of the rift. The user can select each of them and complete the level.

The intermediate sequent $\Gamma, (A \vee B) \wedge C, (\neg B \vee C) \vdash B \vee C, \Delta$ modelled as different bodies in space in the paper prototype. The sequent's solution is explained in section 2.2.2.

Figure 4.2: The paper prototype modelled after an intermediate proof.

The test had three parts. The first part was to directly observe the gestures which were used when interacting with the elements. The users were asked to split a force field, stop a rotation, and use a black hole if there were any. Each move was documented and compiled in a table. By observing the first choice of gesture for each action, it was possible to make a calculated guess of which gestures were more intuitive than others.

After performing their gestures, the users were introduced to the second part of the test: solving proofs. They were presented the simple example $\Gamma, A, (\neg A \vee B) \vdash B, \Delta$, shown in section 2.2.1, with the paper prototype modelled accordingly (see figure 4.1). This example allowed the user to ask inquire hints from the "computer", i.e. one of the team members.

The second proof was more difficult, $\Gamma, (A \vee B) \wedge C, (\neg B \vee C) \vdash B \vee C, \Delta$, shown in section 2.2.2 (see figure 4.2), and the user was asked to solve it on their own. All of their actions were once again written down and compiled in a table.

Lastly, in the third part of the test, the participants were asked to fill out a survey with the following questions:

- **Did you find it easy to understand what you could/could not interact with? Why/why not?** *[Free-form answer]*
- **Is there an operation you would like to change or improve upon? E.g an alternative motion for an operation.** *[Free-form answer]*
- **Was the end goal of the level clear to you?** *[Yes/No]*
- **Did you find a clear path to the goal from the get-go, or did you look for**

possible solutions as you played along? *[Clear path from the get-go/Find it as you played along]*

The results were compiled and used to help in finding out whether the visualisation was playable or not. Furthermore, the gestures which were suggested helped in deciding which gestures were to be used for interacting with the game.

4.4.3 Designing the Game Assets

Since the space-theme could visually represent the sequents, and the test users understood the game, it was time to design the corresponding game assets. First, a colour-theme was selected to make the theme look cohesive. Then, the constituents were created using a vector-drawing application and exported as scalable vector graphics, so that they would always look sharp in any resolution.

5 Developing the Application

Along with defining and choosing a visualisation, developing the application had a top priority. In this chapter, the complete development process is presented. First, the modelling stage of the logic module is presented, where a domain model, a use case model and a class diagram are shown. Then, the implementation stage is introduced, where the choice of platform, the applied software engineering principles and design patterns are presented. Furthermore, the implementation stage also introduces the applied development methodologies and the process of defining the project structures. Lastly, the implementation stage is ended with the process of developing the front-end and connecting it with the logic module.

5.1 Creating Different Models of the Domain

As an initial approach to tackle the problem of constructing a game, the methods and processes of *domain-driven design* were deemed to be the most well-suited for the problem. While there are other applicable methods to the development process, the creation of a domain model, a use-case model and a class diagram were the only models which were created. By first defining the problem domain, a shared understanding would be established within the team. Next, with this understanding, potential use-cases can be defined and explored to find the necessary features for a minimum viable product (MVP). Then, with a scope defined for the MVP, the domain model can be transformed into a class diagram representing the implementation and that fulfils the requirements of the MVP. With these three models, the team deemed it possible to implement most, if not all, requirements for the MVP. For this reason, no other models were determined to be necessary.

5.1.1 Domain Model – Concretising the Problem

A domain model was created to establish a clear understanding of how terms and concepts related to each other. At first, the domain model had a basic structure. However, over time it was further developed and extended in conjunction with brainstorming- and discussion sessions.

Having a domain model is useful in three ways. First, it gives the team a clearer and more general view of the problem domain by mapping its key concepts and their relations. Second, defining the key concepts also forms a domain language of which both domain owners and the team can converse with, which will help to avoid misconceptions. Third, having a well-defined domain model mitigates the risk of designing structures or behaviours that contradicts the conceptual model, as the problem and its related concepts are well understood. This is valuable when composing more complex and specific models, such as the class diagram described in section 5.1.3. The latest revision of the domain model is shown below in figure 5.1.

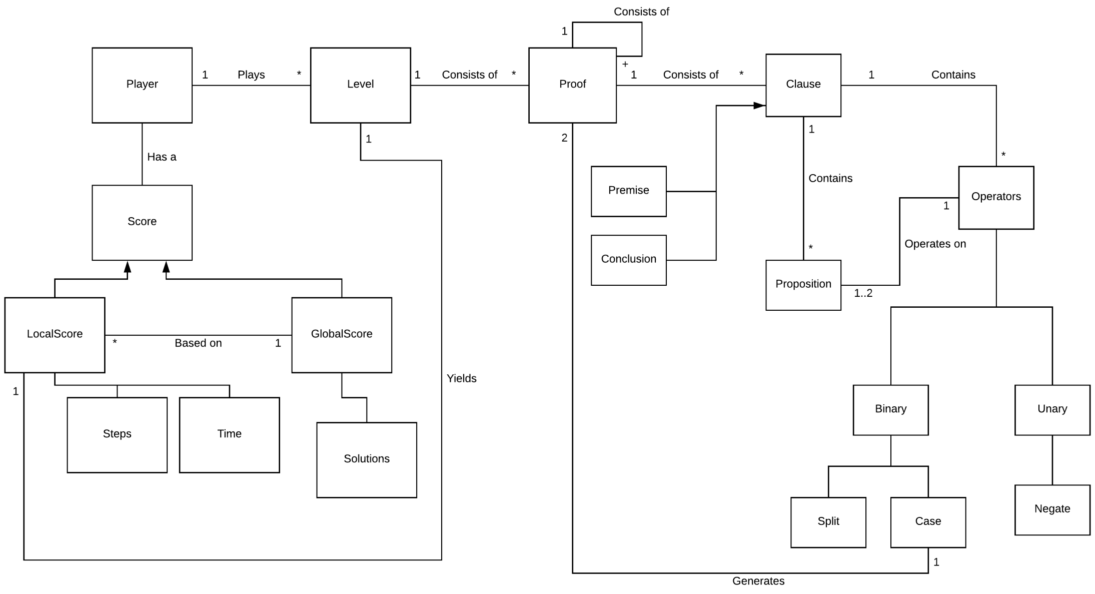


Figure 5.1: Domain model of the problem domain

5.1.2 Use Case Model - Scoping and Defining a Minimum Viable Product

Once the important concepts of the problem domain had been mapped out in the domain model, a use case model was created. This model is made as a set of flows that shows how the user can interact with the application. It also lists what features are necessary for the user to have the intended experience. Creating a use case model early in the development process helped the team to develop code of value as it enforced a focus on the user throughout the implementation. Additionally, a use case model is a useful tool for deciding which features are relevant and in what order they should be implemented. Out of all defined use cases, the ones that were deemed necessary for an MVP were:

- The player interacts with a “Negation” operator.
- The player interacts with a “Disjunction” operator.
- The player interacts with a “Conjunction” operator.
- The player verifies a correct proof.
- The player verifies an incorrect proof.

Further details and the complete use case model can be found in appendix B.

5.1.3 Class Diagram - Transforming the Domain Model and Use Case Model into an Implementable Representation

Along with the domain model and the use case model, a class diagram was created to concretise the problem domain into a structure that is easy to implement. By expanding upon the important concepts and features that were defined in the previous models, the class diagram aims to display a software representation of the problem domain using classes, variables and methods. The class diagram acts as both a blueprint for development and a specification of the program. Initially, the model was created on the whiteboard with the whole team. However, in order to reflect the changes made to solve problems that emerged during the implementation, the class diagram was subject to constant revision throughout the development process. The latest version of the class diagram for the logic module is shown in figure B.1 in appendix B.3.

5.2 Going from Theory to Prototype Through Implementation

With the first iteration of the class diagram model being done, it was time to transform the model into actual code. However, to be efficient with the project early on and to be more familiar with the problem domain, decisions regarding the implementation were deliberately postponed until they were necessary. Doing this allowed the team to make more well-reasoned decisions regarding programming language, game development framework, programming paradigm, software development practices and project structure. This decision process is described in the following subsections.

5.2.1 Choosing a Programming Language, Game Development Framework and Programming Paradigm

During the planning and requirement-definition stage, every member of the team was tasked with researching programming languages and frameworks to use for the application. However, not any programming language would suffice, as the following requirements had to be fulfilled.

Firstly, as the application is a smartphone game, the language had to be compatible with at least one of the platforms Android or iOS. Android supports *Java* or *Kotlin*, and iOS supports *Objective C* or *Swift*. Furthermore, having the ability to develop for additional platforms, and doing so with isomorphic code, would be a strong positive.

Secondly, the language had to be somewhat similar to the languages and paradigms that the team was comfortable with. The languages of which the team had an intermediate, or above, experience level with were *Java*, *JavaScript* and *Haskell*, and the paradigm would therefore have to be either *object-oriented programming* (OOP) or *functional programming* (FP), with OOP being the paradigm of most comfort. Furthermore, having little to no experience in the chosen language would be considered a positive, since all team members looked forward to trying out a new language.

Lastly, a beginner-friendly framework for smartphone game development in the language of choice was of high priority, as this would streamline the development process by not forcing the team to re-invent the wheel of developing smartphone games. Furthermore, if the framework were to be widespread and have a large community, it would be highly probable for it to have many useful libraries. It would also increase the chances of the framework having good support by individuals of the community, which would streamline the development process further and mitigate the risk of stagnating for longer periods of time.

Taking these requirements into account resulted in three options: Kotlin, Swift and C#. After a bit of research, Kotlin was discarded due to the lack of a complete framework for game development when compared to the other options. While Swift had multiple frameworks available, C# had the biggest framework, and game engine, Unity [22]. According to its website, the framework is used to create half of the world's games and is compatible with more than 25 different platforms, including both Android and iOS [22].

Furthermore, C# is an object-oriented, modern and type-safe programming language developed by Microsoft and has its roots in the C family of programming languages [23]. Even though C# is primarily object-oriented, it also provides some functional elements.

With this information in mind, the team quickly reached a consensus: C# and Unity would be the language and framework of choice. Additionally, with Unity having a huge community and a plethora of resources, choosing any other option would result in some form of net loss. Be it the platform support, the ability to get help, or the strong connection with Java (the team's most comfortable language), this combination fulfils all the requirements.

5.2.2 Applied Principles of Software Engineering

When it comes to engineering software, there exist many well-established and thoroughly utilised design principles. Out of the plethora of principles that exist, the principles that were applied in the development process of this project were the following, with the motivation that the team had the highest familiarity with these principles from previous courses:

- The *S.O.L.I.D* principles, found in section A.2.
- *Composition Over Inheritance*, found in section A.3.1.
- *Law of Demeter*, found in section A.3.2.
- *Separation of Concerns*, found in section A.3.3.
- *High Cohesion, Low Coupling*, found in section A.3.4.

- *Command Query Separation*, found in section A.3.5.

By applying these principles early on, preferably even in the modelling stage (if applicable), the probability of producing high-quality code increases. Code of *high quality* is code that has the properties of being *readable*, *maintainable*, *comprehensible*, *structured*, *testable*, *extensible* and *correct*, among others [24, Tab. 2]. A code base of high quality is therefore more inclined to live longer without "going stale" and eventually die off, making it possible to reuse and extend in other applications. The decision was made that the properties mentioned above were to be the indicators for determining the quality of the code base. I.e. the other factors defining code quality were exempt as the team decided that these properties were sufficient for the scope of the project.

Before the implementation could begin, multiple decisions had to be made in order to produce quality code. Firstly, a simple coding standard was defined based on Microsoft's coding standards for C#. The coding standard involves conventions regarding topics such as naming, typing and usage of different constructs. For example, a class or method should be named using `PascalCase`, interfaces should be prefixed with an "`I`" to signify an interface and should always contain at least one member, and fields that could be valuable for extending classes should have an accessibility level of at least `protected` [25]. By following the standard, the code is more likely to be readable and comprehensible as it resembles other code in the same language.

When the coding standard had been defined, discussions were held with regards to how the class diagram model was to be implemented, and how the problems which had occurred were to be solved. To ensure that the code quality would be kept high, the other indicators had to be fulfilled as well. In order to fulfil the testability and correctness aspects of the code base, the team decided to make heavy use of the development methodology *test-driven development* (TDD) along with iterative refactoring to keep the code clean, which is discussed in section 5.2.4.

Extensibility itself is difficult to ensure through a specific method, as the main issue with extensibility focuses on the conceptual idea behind a piece of code. As the project was to be written in C#, the code base would naturally be object-oriented, which often involves difficulties when defining relationships between concepts. It is easy to simply extend a class to implement more specific behaviour. However, such an extension tends to lead to fragile code and, if left alone for a long time, a so-called "big ball of mud" (def. Big ball of mud), where the code is too complex to comprehend. If the code becomes unmanageable, it will also be difficult to extend, which is why the team put heavy emphasis on applying the principles *Liskov's Substitution Principle* (LSP) (section A.2.3), *Open Closed Principle* (OCP) (section A.2.2), *Dependency Inversion Principle* (DIP) (section A.2.5) and *Composition Over Inheritance* (COI) (section A.3.1 when faced with difficult relational problems.

LSP describes what criteria there are for a type to be a subtype of another, and if these are not met, either *polymorphism* (def. Polymorphism) or *composition* (def. Composition) should be used instead. OCP is one of the most general principles, and simply states that code should be "open for extension, closed for modification", i.e. it should be extensible without having to modify the already existing code. COI states that composition should always be used instead of inheritance if composition solves the problem at hand, which acts as an extension of LSP. DIP states that code should only depend on *abstractions* instead of depending on concrete implementation, allowing for concrete implementations to easily be replaced by other implementations, as long as they fulfil the contract defined by the abstraction, which in this case is the interface.

Maintainability and structure of the code base is partly the result of fulfilling the other indicators through the methods previously mentioned. However, when difficult problems occur, finding a solution of high quality may be even harder. Therefore, the decision was taken to make use of design patterns (DPs) when applicable. Introducing them when appropriate had a positive impact on some of the metrics for high-quality code mentioned above. More on the application of DPs is described in section 5.2.3.

5.2.3 Maintainability and Readability with Design Patterns

Design patterns have been mentioned previously, but no real definition has been given so far. Simply put, design patterns are general and repeatable solutions to re-occurring problems in software engineering. Furthermore, incorporating design patterns into the software can, with proper use, be a way to improve the maintainability, structure and readability of the code by using well-known solutions to common problems. They increase the maintainability of the code base by being well-founded solutions to the problems of which they are applied to. Furthermore, DPs act as templates for code structure and are well-known by the software engineering community, which trivially improves the structure and readability of the code.

However, applying DPs where they are not applicable can make the code confusing rather than comprehensible. Since the usage of a specific DP comes with certain expectations of the behaviour and/or structure of that code, misusing DPs might give readers an incorrect impression of the code's purpose or functionality. Which in turn reduces the readability and maintainability, rather than increasing it.

In this project, DPs have mainly been used where they have been blatantly applicable to reap the aforementioned benefits. Examples of applied DPs are the patterns: *abstract factory*, *composite*, and *facade*, all of which are described in section A.1 in appendix A. Furthermore, DPs have been used as a resource that can be drawn upon for inspiration when the team has been faced with problems without obvious solutions.

5.2.4 Applied Development Methodologies

Besides the use of design patterns, the team also applied a few development methodologies, partly because the goal to produce high-quality code remained, but also to make the development process more structured and streamlined. Test-driven development (TDD) was the main focus, as this methodology brings additional good practices into the process. Along with TDD, the agile software development technique Pair programming (PP) was heavily used. Lastly, the methodology of using immutable data structures throughout the code base was applied. All of these methodologies are briefly described and justified throughout the rest of this section.

One example of what additional practices TDD brings to the table is the iterative refactoring process that comes as a result of performing the "red-green-refactor" cycle [26]. During a cycle, the developer first has to write a test for the soon-to-be feature, which naturally will fail. Next, the developer has to write the feature such that the test passes. Finally, when the test passes, the developer has to refactor the code they have written, such that it follows good development principles. Each cycle is quite short and has clear guidelines, which makes the development process more streamlined. and thus resulting in high-quality code.

PP is the process of writing code is always done in small teams of at least two people. While this may seem minor, and possibly inefficient, the benefits of using PP are numerous. An obvious benefit is the ability to discuss and use different perspectives while coding, increasing the chances of finding solutions to problems and catching errors. Another benefit, which may be less obvious, is that PP enforces cooperation and advocates team-wide awareness of a software's different segments. Not only does this increase the bus factor (def. Bus factor), it also makes it more likely to write coherent and comprehensible code which potentially decreases the time spent on merging different parts of a project.

According to the definition above, high-quality code should fulfil the criteria of being testable and maintainable. A key aspect of fulfilling these criteria is creating predictable code. Predictability entails fewer obscure edge-cases, meaning that fewer tests are necessary to exhaustively test the functionality of the code. Moreover, a predictable code is easier to analyse and model, resulting in a decreased risk of creating unsolicited effects when refactoring or extending the code. Immutability is an effective tool for achieving predictability since immutable data structures are exempt from problems associated with aliasing (def. Aliasing). However, the usage of immutable data structures requires more resources due to the constant creation of copies when trying to mutate with a data structure. Therefore, it becomes a judgement call of whether increased performance or increased predictability is more desirable. In this project the team deemed it to be an advantageous trade-off, mainly because performance issues were assessed to be insignificant given the application's resource requirements of a smartphone device.

5.2.5 Defining the Project Structure

Having a well-reasoned project structure plays a major part in fulfilling the criteria of high-quality code. Hastily designed project structures tend to create undesirable dependencies throughout the project, which in turn result in namespaces being highly coupled with each other. High coupling is not desirable, since it may result in a "big ball of mud" if left unattended and thus disallow code reuse and breaking OCP. Instead, by letting the project structure be split up into smaller and highly cohesive assemblies and namespaces, non-related code can be decoupled. Thus, only the assemblies or namespaces, which depend on other assemblies or namespaces, will have to be coupled. Approaching the project in this manner adheres to the *High Cohesion, Low Coupling*-principle.

During the modelling and implementation phases, the team considered this principle on two levels. First, the decoupling of the core logic and graphical representation on the assembly level. Performing this decoupling resulted in the `Core`- and `Game` assemblies, making the non-public code within an assembly invisible to foreign code and thus having high cohesion but low coupling. Furthermore, by having decoupling on the assembly-level, the goal of having a platform-agnostic backend (section 1.3.1) is fulfilled as the `Core` assembly is fully functional on its own and has no external dependencies.

Second, the decoupling of the internal code structures of an assembly by using namespaces. In the case of this project, the namespacing was primarily considered inside the `Core` assembly, as the `Game` assembly heavily depended on Unity and therefore followed the framework's own best practices.

5.2.6 Connecting the Logic Module with Unity

In the meanwhile, as the logic module was close to being finished, a project was set up in the game engine Unity. The assets which were designed from the visualisation phase, found under section 4.4.3, were imported into the project. Several scenes were created along with `GameObjects`, after following the tutorials provided by Unity. Afterwards, scripts were developed to add interactive behaviour to the `GameObjects` in the scenes.

When the logic module was finished, it was built and exported as a `.dll`-file. This `.dll`-file was imported as a library into the Unity project. However, connecting the `Core` assembly with the scenes and the `GameObjects` was not as straightforward as initially thought. While the scenes and scripts existed, it was unclear how to connect them to the provided backend. A few days were spent on trying to figure out how to connect the project from Unity with the `Core` assembly, but the time for the project came to an end.

It is also worth noting that despite the good practices described in section 5.2.4, they were not of focus when working with Unity, partly due to the logic module being the code that would be reused in the future, but primarily as the main focus was figuring out how the game engine worked.

6
Visualisations of Sequent Calculus

One of the main problems defined in this paper was creating a visual representation of the chosen logic. In order for the game to function based on propositional logic expressed in sequent calculus, the following seven elements needed to be visually represented:

- The atomic proposition
- Left-conjunction of propositions
- Right-conjunction of propositions
- Left-disjunction of propositions
- Right-disjunction of propositions
- Negation of propositions
- The turnstile

Furthermore, the player's interaction with the propositions also had to be considered when creating the visualisation. Chapter 4 describes the experimentation and tests that were done in order to find solutions to the stated problems, and the results from them are presented in the following sections.

However, it should be noted that the card game visualisation in section 4.1 is not described in this chapter as it was discarded early on. Thus, neither mockups nor graphics were created for it.

In the sections below, the results from creating different visualisations of the problem domain are presented. First, the second visualisation is presented. Then, aspects which occurred in the early stages of the third visualisation are described, regarding the elimination rules of the chosen operators. Lastly, the third visualisation is introduced, which includes the results from the performed user tests and the final design decisions of this visualisation.

6.1 Visualisation II: Geometric Shapes and Colours

Out of the three visualisations, the second resulted in representing the molecular propositions as basic geometric shapes and colours, which is further described in section 4.2. An atomic proposition would only be a geometric shape consisting of a solid colour. For example, the atomic proposition A is a red circle, and the atomic proposition B is a blue circle. A conjunction of A and B would be represented by a circle where one half is red (A), and the other half is blue (B). A disjunction of A and B would be a circle with stripes of red and blue, instead.

Meanwhile, a negation was represented by a negative space within the geometric shape: $\neg A$ would be a red circle with a hole in it. See figure 6.1 for the described examples.

As mentioned in section 4.2, this visualisation was discarded since it was discovered that the visualisation had issues when it was applied to more complex propositions, such as $(\neg A \wedge B) \vee C$. The attempted visual representation was deemed too complex to interact with, as can be seen in figure 6.2. Therefore, the visual representation for the turnstile, the eliminations of the connectives, and their corresponding gestures were not considered.

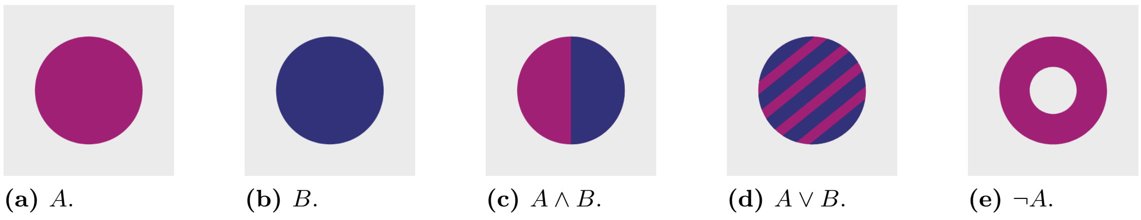
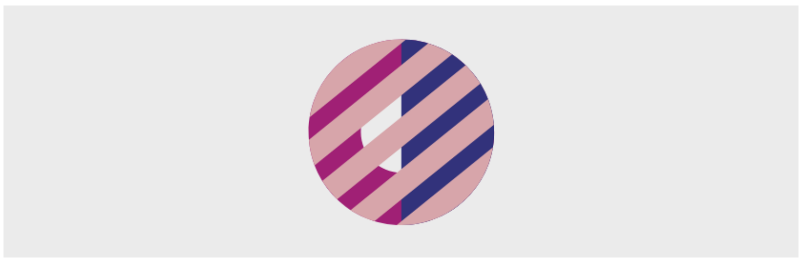

(a) A. **(b)** B. **(c)** $A \wedge B$. **(d)** $A \vee B$. **(e)** $\neg A$.

Figure 6.1: Different visual representations of atomic and molecular propositions in Visualisation II.



The molecular proposition $(\neg A \wedge B) \vee C$ represented in Visualisation II. The dark pink colour represents A, blue represents B, and light pink represents C.

Figure 6.2: A complex visual representation of a molecular proposition in Visualisation II.

6.2 Eliminating the Connectives

As written above, one aspect that needed to be considered was the player's interaction with the propositions. Eliminating a left-conjunction (def. Left/right conjunction) and a right-conjunction had different effects (see explanation in section 2.2). The elimination of a disjunction on the left- or right side of the turnstile were also different.

However, it was found that the elimination of left-conjunctions and right-disjunctions had similar effects (see table 2.2 the similarities). This similarity resulted in left-disjunctions and right-conjunctions sharing the same visual representation. This type of representation was named “split”, since eliminating the operator would simply split the proposition into its constituting propositions.

Similarly, the effect from applying the elimination rule for a right-conjunction and left-disjunction is also the same. As seen in table 2.2, eliminating these operators would result in two new proofs. Each of these proofs replace the operator with one of the two contained operands, which then needs to be solved. Thus, propositions with right-conjunctions and propositions with left-disjunctions share the same visual representations. This type was named “case”, since the player will have to prove each case of the proposition’s operands.

By grouping these operators together according to their elimination rules, the number of visualisations needed were reduced from four to only *split* and *case*.

Lastly, the elimination rule of negation has the same effect on both sides of the turnstile: the contained proposition is moved to the opposite side of the turnstile. Thus, negated propositions are visually represented the same, regardless of where they appear.

6.3 Visualisation III: Space-Theme

After the shortcomings of the previous attempts, this space-themed visualisation aims to solve these shortcomings and is the third and final visual representation. Given the space context, the atomic propositions are each represented by a unique planet.

To represent a split, two propositions are grouped within a *force-field* (a transparent ellipse surrounding the two propositions). Furthermore, a case is represented with two propositions orbiting around a centre point.

A negation is represented by having a black hole behind the negated proposition. Its function is to show that the proposition can be moved into the black hole and use it as wormhole for the proposition to turn up on the other side of the turnstile. The turnstile is a wavy line in the middle of the screen and, within the context of the game, is considered as a *rift* in spacetime.

6.3.1 User Tests

Table 6.1 shows that the gestures used to eliminate the connectives were mixed, and there was no obvious gesture for any of the eliminations. However, some gestures occurred more frequently than others.

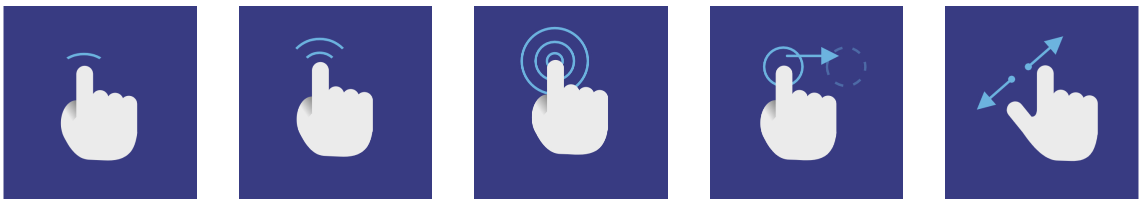
The results from the first user test conducted show that the gestures used to:

- Eliminate a force-field could be a double-tap (3 cases), or drag the planets individually away from the force-field (3 cases), or a reverse pinch on the force-field (1 case).

- Stop a rotation could be a long-press is most frequent (3 cases).
- To use a black hole, a drag-and-drop is the most frequent (5 cases).

Table 6.1: Results from the user tests showing which gestures were used to eliminate the force-fields (splits), rotations (cases), and black holes (negations). The depiction of the gestures are shown in figure 6.3.

#	**Force-Field**	**Rotation**	**Black Hole**
1	Reverse pinch	Reverse pinch	Drag and drop into the black hole
2	Drag each planet out of the force-field	Long-press the area of rotation	Drag and drop into the black hole
3	Drag each planet out of the force-field	Drag the planets out of rotation	Tap on the planet, and then tap on the black hole
4	Double-tap the force-field	Long-press the rotation with two fingers, and then reverse pinch to separate the propositions	Swipe the planet into the black hole
5	Double-tap the force-field	Long-press the area of rotation	Drag and drop into the black hole
6	Drag each planet out of the force-field	Long-press the area of rotation	Drag and drop into the black hole
7	Double-tap the force-field	Pin down the bodies with a responding finger	Drag and drop into the black hole



(a) Single tap **(b)** Double-tap **(c)** Long-press **(d)** Drag and drop **(e)** Reverse pinch

Figure 6.3: Different gestures suggested by the test users.

The other part of the user test where the subjects were instructed to solve a simple proof at first, and then an intermediate proof, revealed interesting points which were not previously considered. Solving the simple proof was straightforward for the majority of the subjects, with the exception of two who were slightly confused.

However, the intermediate proof containing more elements was not as straightforward. Even though all of the subjects managed to solve the intermediate proof in the end, the team members had to interfere at times to explain that some moves were not allowed. For example, some of the subjects tried to stop a rotation while the rotation of bodies were enclosed in a surrounding force field. Another problem was wanting to select a planet to solve the proof, despite it being in a force field or a rotation.

To finish the session, the subjects filled out the survey described in section 4.4.2. The results for the first question show that the majority of the subjects found it easy to understand what they could and could not interact with (6 out 7). One felt it was complicated because there were too many elements on the screen, so they felt overwhelmed and did not know what to interact with. The subjects also felt that after playing the game, the gestures they chose were good as is. For the remaining two multiple choice questions, all of the users unanimously agreed that the end goal was clear to them, and the majority (5) found possible solutions along the way, whereas 2 of the players had a clear path from the beginning.

6.3.2 Visualisation III as the Final Visualisation

The implemented visualisation in Unity follow the space-theme defined in section 6.3. At the top of this chapter, it was pointed out that seven elements had to be visually represented. However, it was shown that split, case and negation would be sufficient to represent all molecular propositions. The elements, now reduced to five, and their representations can be found in table 6.2. To interact with these elements, the gestures chosen are presented in table 6.3.

Table 6.2: Final visual representations of the elements needed.

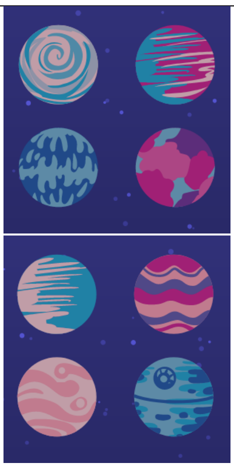
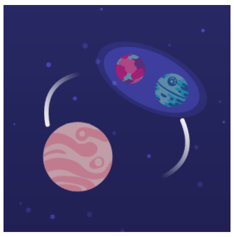
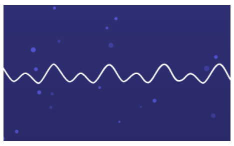
Element	Representation	Designed Assets
Atomic proposition	Free planet.	
Split	Two molecular propositions within a force field.	
Case	Two molecular propositions rotating around each other.	
Negation	A black hole behind an atomic or a molecular proposition	
Turnstile	A rift in spacetime, depicted as a wiggly line across half the screen	

Table 6.3: Final gestures chosen to interact with the elements.

Element	Gesture
Selecting a planet	Tap the planet.
Splitting a force field (split)	Double tap the force field.
Stopping a rotation (case)	Long press the area of rotation.
Using a black hole (negation)	Drag and drop the atomic or molecular proposition into the black hole

7

Tenjin Core: The Brains of the Operation

Out of the two C# assemblies (see section 2.3.1) for *Tenjin*, *Tenjin Core* is the assembly which contains the implemented representation of sequent calculus for propositional logic. Within this assembly there are two major namespaces (see section 2.3.1), the `Game` namespace and the `Logic` namespace.

7.1 Manipulating Proofs Using the `Game` Namespace

The `Game` namespace contains the logical representation of a "level" in the game along with the tools to interact with that level. More precisely, the `Level` class manipulates the currently encapsulated proof and keeps track of all resulting subgoals that need to be solved to complete the proof and finish the level. This class also tracks metrics, such as the number of performed moves and a timer. These metrics are to be used when calculating the final score when finishing a level. To interact with a `Level`, an instance of a `ILogicOperator` is necessary. For the case of this project, the provided implementing class `PropositionalOperator` is sufficient. Furthermore, for code that is foreign to this assembly, the main entry point to handle and modify proofs is through the `ILevel` interface, which `Level` implements and `ILogicOperator` fully depends upon.

7.2 Constructing Proofs Using the `Logic` Namespace

In contrast to the `Game` namespace, the `Logic` namespace is intended to fully model proofs according to the underlying domain. In order for the representation to be accurate and to remain extensible, the majority of the namespace contains little-to-no behaviour. Instead, it primarily serves a structural purpose. The behavioural logic is instead delegated to the aforementioned `PropositionalOperator`. Inside the namespace, another namespace called `Elements` exists which contains interfaces, a factory, and classes for atomic propositions and the molecular propositions negation, disjunction and conjunction. The interfaces are what the `ILogicOperator` interface depends upon, so no namespace knows more than necessary about other namespaces' concrete implementations. Additionally, the interfaces in the `Elements` namespace allow foreign code to implement external concrete types. The types are fully compatible with the Core assembly, making the assembly highly reusable.

8

Tenjin as the Resulting Application

`Tenjin Game` is the assembly which contains the implemented visualisation of `Tenjin Core`. In this chapter, the resulting application is presentend. It is composed of the elements provided by Unity, coupled with scripts written in C# to define the behaviour of the game objects related to the `Core` assembly. The following sections describe the how the `Core` assembly is visualised, how proofs work within the game and how the different elements of the visualisation are implemented in Unity.

8.1 Visualising `Tenjin Core` with Unity

Making use of the `Tenjin Core`-model's namespaces `Game`, `Logic`, and `Elements`, `Tenjin Game` ties a visualisation to the model, without introducing additional dependencies. The main script is `ProofScript.cs`, where methods are constructed to model the proofs visually. The methods `Start, Awake` and `Update` are the basic functions of the script. These methods control initiation and updates of visual objects and effects.

Designed by the hand of a team member, the assets for objects and backgrounds are used to visualise the game elements. Methods which are defined in various scripts then depend on these assets, instantiating objects and using them as skins.

8.2 Proofs in the Game

While the behaviour of the elimination rules applied to proofs is defined in `Tenjin Core`, the visual effects and representations are handled by the scripts. Using the `Planet` class as a container class for atomic propositions of the model, referred to as `IAtomicProposition`, the proofs are initialised by the method `DrawProof`. This method uses the overloaded method `DrawPlanet` to visualise different premises and conclusions in planet form.

As the game is updated by the `Update`-method every frame, the consequences from modifying a proof are displayed. Each time a proof changes, it gets redrawn by calling the `DrawProof`-method once again. When the current proof is confirmed to be proven, the next proof of the current level is drawn.

8.3 The Representation of Planets and Other `GameObjects`

In this section, the implementation of the visual elements and game objects are described. First, the rift-object is introduced. Next, the planet- and black hole-object are presented. Finally, the force-field- and rotation-object are described.

8.3.1 The Rift in Space

As mentioned in 6.3, the rift represents the turnstile. Premises are listed on the top of the rift and conclusions on the bottom. The rifts behaviour is defined in `DividerScript.cs`, where the drawn divider between the two spaces is composed of 100 vertices constantly updating in the `Update`-method. The form of this moving rift is described by a position, an amplitude, a speed-factor and a `LineRenderer` in Unity.

8.3.2 The Free Planets and Black Holes

Since the atomic propositions are displayed as free planets, they are visually constructed by attaching a planet-asset to a `GameObject`. By making use of the predefined Unity-method `Instantiate`, objects are instantiated visually. Instantiation is done by providing characteristic arguments such as position, model and rotation.

The black hole is an effect which can be applied to game objects, indicating that a planet or a compound of planets can be moved across the rift in space, if the black hole is interacted with. The black hole-object is instantiated and defined as a child-object to the planet that it belongs to.

8.3.3 The Force-fields and Rotations

Compounds of propositions are commonly interacted with during proof elimination as the goal is often to reach one of the inner planets. As for implementation, force-fields and rotations are implemented similarly. Three `GameObjects` form a compound, with an invisible parent-object defining the behaviour of two child-objects. These child-objects are in the simplest form two planets, but there are also scenarios where force-fields, black holes and rotations are interchained.

To define the scenario of a *split* in a proof, two propositions are put in a force-field. By using an asset that covers the two children-objects, the parent-object produces a force-field-effect.

As for the *case*-scenario, it is instead a rotation consisting of two propositions rotating around an invisible parent-object using the method `transform.Rotate` for `GameObjects`. This behaviour is mainly defined by the overloaded method `DrawPlanet`, located in `ProofScript.cs`.

9

Discussion

By taking the results that have been presented in the previous chapters, and the methods leading to these results, this chapter aims to critically analyse and justify the intent of this project. First, it compares the initial expectations of the project with the end-results and highlights the possible reasons behind any deviations. Next, it discusses some of the methodologies that have been applied throughout the project and proposes possible alternatives that could have made the process easier or more efficient. Finally, it presents what remains to be done with Tenjin and proposes topics for future work based on this topic.

9.1 Expectations and Final Result of the Application

Comparing the criteria for a successful application found in section 1.3.1 with the developed application, some aspects of the produced results excelled, whereas others fell short of the expectations.

Firstly, the minimum number of visualisations were fulfilled according to the fourth criterion, where the team produced the three themes: card game-theme, geometric shapes and colours-theme, and space-theme. However, only the space-theme got into the prototyping stage and was tested, which makes it arguable whether the first two themes can be viewed as "alternative visualisations" or not. Even though the team set out to create paper prototypes and to perform user tests for all visualisations, the time it took to create a single paper prototype and test it made it apparent that repeating the process twice more would take too much time out of the project. Conversely, when looking back, the effort that was put into the paper prototype could have been distributed between the three visualisations, as the space theme prototype is over-designed for something that is supposed to be thrown away. By doing this, user tests could have been performed on all three visualisations and more diverse testing data could have been accumulated. Nonetheless, the results from testing the other visualisations could be questionable, as the team noticed problems with the first two visualisations during their respective design stage.

Secondly, the front-end of the application, and therefore the game itself, is not developed to the stage of being a complete experience. Therefore, the third criterion in section 1.3.1 regarding a fully implemented visualisation is not fulfilled. Currently, the application only has predefined levels with different proofs that are randomly chosen on either application launch or level completion. However, there are no menus, level selection, nor a complete set of animations in the level view. Additionally, the different gestures that were supposed to be implemented for each action, shown in table 6.3, are exempt, mainly due to the gestures not having a high priority compared to visualising a level in the front-end.

There are two primary reasons for the application's missing features. One is the lack of planning and modelling of the front-end compared to the backend, as well as the communication between them. The other is the lack of knowledge as to the preferred workflow of the Unity framework. The former reason partly originates from the latter, since the modelling of the front-end was deferred until the team had a firmer grasp on the development flow in Unity. However, as learning the development flow of Unity became a greater task than initially expected, no in-depth planning nor any modelling for the front-end was made. Furthermore, due to both .NET and Unity having little-to-no documentation regarding the requirements of how to build, export and import assemblies of the correct format, the procedure of integrating the backend into the Unity project had to be done through trial-and-error, which resulted in a time sink.

Thirdly, the backend of the application is fully platform-agnostic, and can be run on any C# runtime that supports .NET Standard 2.0 (see section 2.3.2). Furthermore, as the structural and behavioural logic are separated from each other, the `Core` assembly can easily be extended to include additional logical operators, e.g. implication or biconditional implication (see table 2.1). The separation of structure and behaviour also open up for other logic of the zeroth order, e.g. zeroth order constructive logic or zeroth order temporal logic, by implementing the `LogicalOperator` interface. In regards to the first criterion in section 1.3.1, the `Core` assembly exceeds the expectations.

However, the second criterion regarding proof generation is unfortunately not fulfilled. The consensus was that for this feature to be taken to the implementation stage, the MVP had to be finished. Further discussion of this feature and how it can be developed is found in section 9.2.

9.2 Applied Methodologies and Possible Alternatives

The team spent a significant portion of the time to try out the different visualisations, and this is partly due to being unstructured. It was difficult to come up with something that would work immediately for all of the criteria mentioned at the beginning of chapter 6. Therefore, the team went on a trial-and-error route when coming up with ideas, which might not have been the most effective. However, using the benchmark proofs (see section B.4 in appendix B) helped weed out the visualisations that did not work.

Moreover, the team was not sufficiently prepared when trying out the two first visualisations. Intuitive design was not considered at all, so a literature study for visualisations was done after two ideas were already tested. Perhaps it would have been better to start with a literature study regarding the visualisations before starting out on them.

As mentioned in the previous section, the paper prototype which was created was over-designed. To counteract this in the future, paper prototypes could be simple sketches without fillings. However, the immersion factor for the user might disappear if the prototype looks too simple. Another possible solution would be printing images from the computer for the constituents, instead of hand-painting them.

Additionally, the paper prototype was used in a user test, as described in section 4.4.2, and the user test definitely has room for improvements. Even though the users managed to solve the proofs, it was not straightforward at times. The source of confusion could have stemmed from the team members holding the test not being clear enough.

While the team members had a script which they could follow for the user tests, it was neither thorough nor detailed. It listed the essentials which the users were asked to do, but the rules of the games were not described explicitly in the script. Not having the explanation of the rules written down led to variations of the explanation between the different tests. The users from the tests conducted in the beginning had a higher risk of being more confused, as the team members were not well-versed in explaining the rules. However, the confusion of the users helped the team members be more clear in the coming explanations for the next users. To stay clear of this type of variation, and confusion, the scripts for user-tests would need to be more detailed. The rules of the games would be written down, and the team would prepare first by holding a test-round of the user test internally. That would help clear most of the confusion, and not give rise to big variations between the tests.

Continuing on the user tests, there were some interesting results for the gestures suggested (see table 6.1). Some of the gestures required two fingers, which means that the user would have to hold the smartphone in one hand and playing with another, or that their phone would sit on a surface. However, the size of the paper prototype might have affected their choice of gesture, as the paper prototype had the dimensions of an A4-sized paper. It is not clear whether their choices would have remained the same if they were presented with a paper prototype closer to that of a smartphone screen. With a smaller prototype, they might have chosen gestures which only required one finger in order to enable one-hand usage of a smartphone. To improve the user tests, the paper prototype should be developed closer to the actual size of the device that the game is to be played on.

On another point, as described in section 5.2.1, the `Core` assembly was fully written in C#, which entails that the whole assembly is object-oriented. Even though there are some functional elements in the language, these elements are syntactical sugar and get resolved to an object-oriented structure for the runtime. In hindsight, this fact is quite relevant for the representation of sequent calculus, since it meant that the mathematical model of sequent calculus had to be transformed into an object-oriented class hierarchy. Representing sequent calculus using classes and other object-oriented constructs is arguably unnatural, as the relationships between logical connectives and atomic propositions were found to be very difficult to define.

Although the `Core` assembly is representational of the elimination rules for negation, disjunction and conjunction within sequent calculus, the solution that was developed is misusing interfaces at points by leaving them empty as to only give types to depend on which lack concrete implementations. Such an approach is more of a functional programming nature and since sequent calculus, and mathematics in general, is similar to the functional programming paradigm, it is reasonable that the `Core` assembly would be both easier and better to implement in a functional programming language, such as Microsoft's F#. By developing the `Core` assembly in F#, the code would arguably be more readable as it is more similar to the formal syntax of sequent calculus. Furthermore, the code would be less bloated with object-oriented boilerplate code and would still be interoperable with any C# code. Therefore, if it would be simpler to represent sequent calculus in F# compared to C#, then it is possible that the application would have reached a more complete state by the end of the project. However, this is not necessarily true, as writing the `Core` assembly in F# and the `Game` assembly in C# would mean that the team would need to learn two languages instead of one, resulting in additional overhead.

One aspect that should be discussed is that the team did not enforce any internal roles or responsibilities on a specific person. As the team consists of only four people, it felt unnecessary at the time. However, it is interesting to think if the project would have fared off better if there were roles such as a product owner. A product owner could help the team prioritise, and therefore indirectly help keep the deadlines.

While there was no role assigned to a specific person, the division of work between visualisations and coding gradually split the group in two. This allowed parallelism, and more work could be done in each area. However, as can be seen in figure 3.1, the two areas eventually merged back into one. The merging of these two areas did not result in a fully playable game, as described in section 5.2.6, and a possible cause for this could be a lack of communication between the two parties. As the section describes, using the backend was not straightforward for the party that worked with the visualisations. In short, the two parties were not in sync, and to solve this for future projects, the team can implement daily stand-ups. The stand-ups provide an opportunity for each member to tell what they are working on and if they need help. Furthermore, they could also summarise what they have been working on so far, to keep everyone up to date.

In general, the project suffered when the tasks could not be divided so that work could occur in parallel. Such tasks could, for example, be coming up with visualisations, as mentioned above. However, it was imperative that everyone would agree on the visualisation, so there was no way around that. Yet, there was not much progress for a while, and one way to counter the stagnation is to ask for external help. Should the project continue, and the team is stuck on a problem for too long in regards to the planning, the team should ask for help.

9.3 Continuing the Story of Tenjin

In order to bring the current version of *Tenjin* to a fully featured and complete experience, there are some essential aspects that still are necessary to implement in the front-end. The majority of these aspects are related to navigation, such as menus for selecting levels and options. The interactive and visual elements of a level are not fully implemented either, since the chosen gestures described in table 6.1 are not all implemented and some of the animations need some tuning for a more realistic effect.

Furthermore, for the backend to be complete according to the set criteria in section 1.3.1, the ability to generate proofs and therefore generate levels needs to be implemented. During the research stage, the team came into contact with Maximilian Algehed, a PhD at the

Functional Programming division of the Department of Computer Science at Chalmers, who gave the tip of contacting Alex Gerdes, a senior lecturer at the same division, regarding his paper *"Strategies for Exercises"*. In this paper, Gerdes and his co-authors are introducing a language for calculating complete solutions to a problem, given the appropriate strategies [27]. For future work, this would be an interesting point to start from and to see if proofs can be generated using the strategies defined by the elimination rules in sequent calculus.

Leaving behind this project's set criteria, another proposal for future work would be to extend the `Core` assembly. Such an extension could be by implementing another logic, e.g. the logics mentioned in section 9.1, or by implementing introduction rule logic for the defined operators. Additional extensions would be implementing additional operators, e.g. the operators mentioned in section 9.1 above, or to add logic for first or higher order logic, which would allow a plethora of additional logics to implement.

Lastly, for the user experience-interested, there are multiple topics which can be explored. One example is to perform further prototyping and user testing of the proposed space-theme for future work. Additionally, producing an alternative visualisation, or modified versions of the existing ones, to then compare with the space-theme or the other two visualisations in sections 4.1 and 4.2 is also an interesting topic. Furthermore, the process of producing an alternative visualisation could be taken a step further by defining a new key aspect for intuitive visualisations (see section 4.3), and then base the visualisation on this key aspect.

10 Conclusions

Logical reasoning plays an important role, partly in everyday life, but mainly in more educational settings. Unfortunately, since the subject of logic and proofs is primarily taught at university and the learning curve being quite steep, many lack the formal concepts of the subject. This could pose a problem, as it could lead to logical errors, such as logical fallacies. As this project aims to help with this issue through gamification of logic and proofs, it solves two inherent problems.

First, it provides an accurate implementation of propositional logic in the form of sequent calculus, with the limitation of only focusing on classical logic and the elimination rules for the logical operators *negation*, *disjunction*, and *conjunction*. Additionally, the implementation is built using good software engineering practices in such a way that it should be extensible to other logics and additional logical operators.

Second, it proposes a visualisation that is representative of the chosen logic and limitations, which has been user tested for validity. The visualisation is based on the aspect of “naïve physics” and takes the shape of a space-inspired theme. Planets act as atomic propositions and black holes act as instances of the negation operator. Furthermore, bodies rotating around the centre point between them, as well as force fields surrounding different bodies both act as instances of the disjunction and conjunction operators, depending on the context in which they occur. This visualisation has proved to be effective in masking the syntactic notation of logic and proofs and allowed the students to come in contact with the formal concepts without the academic context.

Furthermore, a set of criteria were defined to scope the project and to give context when interpreting the result. Out of these criteria, the first one is fully completed, the second and third criteria have not reached a state where they can be considered fulfilled, and it is arguable whether the fourth one has been fulfilled or not.

Nevertheless, since the proposed visualisation has been shown to work through the user tests, and that the provided implementation is fully functional, this thesis provides useful information regarding both user experience and educational game development. Firstly, defining restrictions as the base for designing a visualisation is preferable as it narrows down the search space of possible options and consequently reduces the time spent on experimenting with inherently flawed alternatives. Secondly, when developing the backend logic for a game that has a well-defined specification of requirements, taking the extra time to create representative models is useful. In the instance of this project, this meant that the implementation of sequent calculus never had to be modified, only extended when there was a need. Finally, this thesis may act as a blueprint for one approach of creating a game that is modelled after formal mathematics.

Moreover, this thesis creates opportunities for future work in continuing building upon this project. For example, the application needs to be completed to result in a game, according to the third criterion. Also, the provided implementation needs to be extended such that proofs can be generated dynamically. Additionally, outside the scope of this project, the implementation could be extended with other logics, additional operators and introduction rules for classical propositional logic. Lastly, additional user testing and prototyping, along with the creation of multiple visualisations are also relevant to find a more intuitive and effective visualisation for learning logic and proofs.

Bibliography

[1] F. Pfenning. (). Philosophy 103: Introduction to logic argumentum ad ignorantiam, [Online]. Available: https://philosophy.lander.edu/logic/ignorance.html (visited on 05/18/2019).

[2] Skolverket. (). Det här gör skolverket, [Online]. Available: https://www.skolverket.se/om-oss/organisation-och-verksamhet/det-har-gor-skolverket (visited on 04/22/2019).

[3] ——, (). Undervisning, [Online]. Available: https://www.skolverket.se/undervisning (visited on 04/22/2019).

[4] ——, (Feb. 13, 2019). Ämne - matematik (grundskolan), [Online]. Available: https://www.skolverket.se/undervisning/grundskolan/laroplan-och-kursplaner-for-grundskolan/laroplan-lgr11-for-grundskolan-samt-for-forskoleklassen-och-fritidshemmet?url=1530314731%2Fcompulsorycw%2Fjsp%2Fsubject.htm%3FsubjectCode%3DGRGRMAT01%26tos%3Dgr&sv.url=12.5dfee44715d35a5cdfa219f.

[5] ——, (Feb. 13, 2019). Ämne - matematik (gymnasieskolan), [Online]. Available: https://www.skolverket.se/undervisning/gymnasieskolan/laroplan-program-och-amnen-i-gymnasieskolan/gymnasieprogrammen/amne?url=1530314731%2Fsyllabuscw%2Fjsp%2Fsubject.htm%3FsubjectCode%3DMAT%26tos%3Dgy&sv.url=12.5dfee44715d35a5cdfa92a3.

[6] ——, (). Ämne - programmering (gymnasieskolan), [Online]. Available: https://www.skolverket.se/undervisning/gymnasieskolan/laroplan-program-och-amnen-i-gymnasieskolan/gymnasieprogrammen/amne?url=1530314731%2Fsyllabuscw%2Fjsp%2Fsubject.htm%3FsubjectCode%3DPRR%26courseCode%3DPRRPRR01%26tos%3Dgy%26p%3Dp (visited on 02/13/2019).

[7] P. D. A. Thoresson, "Svenskarna och internet 2017", IIS, Tech. Rep., Oct. 2017. [Online]. Available: https://internetstiftelsen.se/app/uploads/2019/02/Svenskarna_och_internet_2017.pdf (visited on 02/14/2019).

[8] K. C. Klement. (). Propositional logic, [Online]. Available: https://www.iep.utm.edu/prop-log/#SH3a (visited on 04/18/2019).

[9] J. van Benthem, H. van Ditmarsch, J. van Eijck, and J. Jaspars. (2016). Chapter 2 / propositional logic from logic in action, [Online]. Available: http://logicinaction.org/docs/lia.pdf (visited on 04/01/2019).

[10] G. Gentzen, "Untersuchungen über das logische Schließen", *Mathematische Zeitschrift*, vol. 39, pp. 176–210, 405–431, 1935, English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969. [Online]. Available: https://doi.org/10.1007/BF01201353.

[11] A. Sakharov. (). Sequent calculus, [Online]. Available: http://mathworld.wolfram.com/SequentCalculus.html (visited on 03/29/2019).

[12] D. Baelde. (2014). Classical sequent calculus, [Online]. Available: http://www.lsv.fr/~baelde/1415/logique/lk.pdf (visited on 04/04/2019).

[13] L. University. (). Lecture notes on structural logic, [Online]. Available: https://www.cs.cmu.edu/~fp/courses/15816-f16/lectures/14-structural.pdf (visited on 05/15/2019).

[14] E. Z. Yang. (2012). Interactive tutorial of the sequent calculus, [Online]. Available: http://logitext.mit.edu/tutorial (visited on 04/04/2019).

[15] M. Wenzel, S. Addie, A. Patridge, and T. Sandstrom. (). Dotnet new command, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/tools/dotnet-new?tabs=netcore21 (visited on 05/13/2019).

[16] M. Wenzel, L. Latham, B. Wagner, T. Pratt, R. Petrusha, V. V. Agarwal, M. Jones, P. Carter, R. Lander, Y. Jin, T. Zhang, A. A, and S. Boyer. (2019). .net framework guide, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/index (visited on 04/26/2019).

[17] M. Wenzel, B. Wagner, M. Hoffman, and J. Gérard. (). C# namespace reference, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/namespace (visited on 05/13/2019).

[18] I. Landwerth. (2017). Demystifying .net core and .net standard, [Online]. Available: https://msdn.microsoft.com/en-us/magazine/mt842506.aspx (visited on 04/27/2019).

[19] L. Volk. (2016). Review of the most popular engines for game development, [Online]. Available: https://vironit.com/review-of-the-most-popular-engines-for-game-development/ (visited on 04/12/2019).

[20] U. Technologies. (2019). Game engines—how do they work?, [Online]. Available: https://unity3d.com/what-is-a-game-engine (visited on 04/12/2019).

[21] I. D. Foundation. (). What is intuitive design?, [Online]. Available: https://www.interaction-design.org/literature/topics/intuitive-design (visited on 02/25/2019).

[22] U. Technologies. (). Unity overview, [Online]. Available: https://unity3d.com/unity (visited on 04/30/2019).

[23] B. Wagner, N. Schonning, M. Wenzel, L. Latham, and P. Onderka. (Apr. 2019). A tour of the c# language, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ (visited on 04/30/2019).

[24] J. Börstler, H. Störrle, D. Toll, J. van Assema, R. Duran, S. Hooshangi, J. Jeuring, H. Keuning, C. Kleiner, and B. MacKellar, "I know it when i see it perceptions of code quality: Iticse'17 working group report", in *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, ACM, 2018, pp. 70–85.

[25] M. Wenzel, D. Lee, N. Turn, L. Latham, T. Pratt, R. Petrusha, M. Hoffman, M. Jones, C. Maddock, Y. Jin, and A. A. (). Framework design guidelines, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/ (visited on 04/25/2019).

[26] R. C. Martin. (Dec. 2014). The cycles of tdd, [Online]. Available: https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html (visited on 05/07/2019).

[27] B. Heeren, J. Jeuring, and A. Gerdes, "Strategies for exercises", *Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2009-003*, 2009.

A

Design Patterns and Software Engineering Principles

A.1 Design Patterns

The following design patterns are excerpts from the ebook *Dive Into Design Patterns*, found at https://sourcemaking.com/design-patterns-ebook.

A.1.1 Abstract Factory pattern

More information about the Abstract Factory pattern can be found at https://sourcemaking.com/design_patterns/abstract_factory.

Intent:

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".
- The new operator considered harmful.

Problem: If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulation is not engineered in advance, and lots of `#ifdef` case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

A.1.2 Composite pattern

More information about the Composite pattern can be found at https://sourcemaking.com/design_patterns/composite.

Intent:

- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- "Directories contain entries, each of which could be a directory."
- 1-to-many “has a” up the “is a” hierarchy

Problem: Application needs to manipulate a hierarchical collection of “primitive” and “composite” objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the “type” of each object before attempting to process it is not desirable.

A.1.3 Facade pattern

More information about the Facade pattern can be found at
https://sourcemaking.com/design_patterns/facade.

Intent:

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.

Problem: A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

A.2 S.O.L.I.D Software Design Principles of Object-Oriented Programming

The following information are directly copied off of the site Baeldung, from the page https://www.baeldung.com/solid-principles. More information and clarifying examples can also be found on this page.

The SOLID principles were first conceptualized by Robert C. Martin in his 2000 paper, *Design Principles and Design Patterns*. These concepts were later built upon by Michael Feathers, who introduced us to the SOLID acronym. And in the last 20 years, these 5 principles have revolutionized the world of object-oriented programming, changing the way that we write software.

So, what is SOLID and how does it help us write better code? Simply put, Martin’s and Feathers’ **design principles encourage us to create more maintainable, understandable, and flexible software**. Consequently, **as our applications grow in size,**

we can reduce their complexity and save ourselves a lot of headaches further down the road!

The following 5 concepts make up our SOLID principles:

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

A.2.1 Single Responsibility Principle

Single Responsibility principle states that **a class should only have one responsibility. Furthermore, it should only have one reason to change.**

How does this principle help us to build better software? Let's see a few of its benefits:

- Testing – A class with one responsibility will have far fewer test cases
- Lower coupling – Less functionality in a single class will have fewer dependencies
- Organization – Smaller, well-organized classes are easier to search than monolithic ones

A.2.2 Open Closed Principle

Open closed principle states that **classes should be open for extension, but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs** in an otherwise happy application.

Of course, the **one exception to the rule is when fixing bugs in existing code.**

A.2.3 Liskov's Substitution Principle

Liskov substitution is arguably the most complex of the 5 principles. Simply put, **if class A is a subtype of class B, then we should be able to replace B with A without disrupting the behavior of our program.**

A.2.4 Interface Segregation Principle

Interface segregation principle simply means that **larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

A.2.5 Dependency Inversion Principle

The principle of Dependency Inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.

A.3 General Principles of Object-Oriented Programming

This section covers the principles applied to the development of the application. The following information are directly copied from www.java-design-patterns.com/principles.

However, subsection A.3.4, which has information from slides from the course *Object-oriented programming and design* (DIT952) by Niklas Broberg, Department of Computer Science at Chalmers University of Technology.

A.3.1 Composition over Inheritance

Why? Less coupling between classes. Using inheritance, subclasses easily make assumptions, and break LSP(A.2.3).

How? Test for LSP (substitutability) to decide when to inherit. Compose when there is a "has a" (or "uses a") relationship, inherit when "is a".

A.3.2 Law of Demeter

Don't talk to strangers. It usually tightens coupling. It might reveal too much implementation details.

A method of an object may only call methods of:

- The object itself.
- An argument of the method.
- Any object created within the method.
- Any direct properties/fields of the object.

A.3.3 Separation of Concerns

Separation of concerns is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. For example the business logic of the application is a concern and the user interface is another concern. Changing the user interface should not require changes to business logic and vice versa.

A.3.4 High Cohesion, Low Coupling

Cohesion is a measurement of the inner coherence of a module. High Cohesion means that the modules defined in a program are designed to solve the problem they are assigned to, rather than relying on components in other modules.

Coupling is a measurement of how dependent two different modules are on each other. High coupling would eventually result in that a change in one module forces a change in a module dependent on it, which breaks the Open-Closed Principle described at A.2.2.

A.3.5 Command Query Separation

The Command Query Separation principle states that each method should be either a command that performs an action or a query that returns data to the caller but not both. Asking a question should not modify the answer.

With this principle applied the programmer can code with much more confidence. The query methods can be used anywhere and in any order since they do not mutate the state. With commands one has to be more careful.

B

Project Models and Documents

B.1 Tenjin: Paper Prototype User Test

B.1.1 Ask them how they perform the actions

1. Show them how a split may look like. *How do you separate these planets?* **Tell them what happens after they have split the force field.** (Nothing, they can continue to solve the puzzle.)

2. Show them how a case may look like (the rotation of two bodies). *How do you separate these planets?* **Tell them what happens after they separated the bodies from the orbit.** (Each body will be focused on, respectively, and they can continue to solve the puzzle.)

3. Show them how negation may look like (a black hole behind a body). *How do you move this body to the other side of the rift?* **Tell them what happens when they have moved the body to the other side of the rift.** (The body is on the other side of the rift, and they can continue to solve the puzzle.)

B.1.2 Hold their hands to solve the simple proof

$$\Gamma, A, (\neg A \lor B) \vdash B, \Delta$$

1. Split A and $(\neg A \lor B)$. *(The planets are separated and the force field disappears.)*

2. $(\neg A \lor B)$ from their orbit. *($\neg A$ is focused on, B is moved out of focus.)*

3. Move $\neg A$ to the other side of the rift through the black hole. *(A pops in the black hole and appears on the other side.)*

4. Tap A on both sides of the rift to repair the rift (proving the statements). *(The planets representing A disappear, and we are left with B. The B out of focus moves back in focus.)*

5. Same as (4). *(The user is finished and has proved the statement. They finished the*

level!).

B.1.3 Present a more challenging proof and ask them to solve it

$$\Gamma, (A \lor B) \land C, \neg B \lor C \vdash B \lor C, \Delta$$

B.2 Requirement Analysis Document

B.2.1 Introduction

This project is about creating a puzzle game designed for smartphones. The puzzles should be based on logical proofs and solving a puzzle should be equivalent to mathematically solving the underlying proof.

B.2.2 Functional requirements

- Interact with operators:
  - Negate
  - Split
  - Case
- Verify that a proof has been solved
- Get a hint
- Access menus:
  - Main menu
  - Map
  - Options
  - Continue
  - Leaderboard
- Select level

B.2.3 Non-functional requirements

- Scores from:
  - Single level
  - All levels
- Proof generation
- Usability
- Performance
- Implementation
- Testability
- Persistence
- Legal

B.2.3.1 Proof generation

There should be a possibility to generate proofs, based on parameters such as number of steps needed to solve, for the player to solve.

B.2.3.2 Usability

In order to expose a broader audience to the mathematical solving of logical proofs, extra emphasis should be put on making the game as intuitive as possible.

B.2.3.3 Performance

The game should be optimised for minimal loading times and fast response.

B.2.3.4 Implementation

The application will be written in C# and the visualisation will be written using the framework Unity.

B.2.3.5 Testability

The model should be created using test-driven development and NUnit. The Unity-based parts should have covering tests in order to verify deterministic behaviour.

B.2.3.6 Persistence

The application should store essential data, allowing the player to experience the game throughout multiple playing sessions without losing progress.

B.2.4 Use cases

This section states the use cases defined by the agreed requirements for the application.

B.2.4.1 Interacting with operators

Goal: The user interacts with an operator, using the chosen gesture, to progress in solving the proof.
Priority: High
Actor: User
Description: While playing a level, the user interacts with the top-level operator of a clause. The clause dissolves and the proof is replaced by the next state of the proof, which is dependent on the operator.

Flow: The player interacts with a “negate” operator.

	Actor	System
1	Long press on the negated clause.	
2		A portal appears above the clause.
3	Drags the clause into the portal.	
4		The non-negated clause appears on other side.

Flow: The player interacts with a “Split” operator.

	Actor	System
1	Clicks on “split”-clause.	
2		The clause is replaced by both its contained operands.

Flow: The player interacts with a “Case” operator.

	Actor	System
1	Clicks on “case”-clause.	
2		The clause is dissolved into two new proofs. In the first proofs, the clause is replaced by the first operand and in the second proof the clause is replaced by the second operand .

B.2.4.2 Verify proof

Goal: Verify a proof to see if the user has passed the level or not. The system responds with either a score for the finished level, or indicates an error and accompanied penalties.
Priority: High
Actor: User
Description: While playing a level, the user selects a proposition on one side of the proof. Then, the user selects another on the opposite side. The system tries to verify the proof. If the two propositions match, the proof has been verified and the user is awarded their level score. Otherwise, the user is informed that the proof was incorrect. When the user then finishes the level, the system will penalise the user by deducting points off of the final score.

Flow: The player verifies a correct proof.

	Actor	System
1	Taps on one proposition on one side.	
2		Highlights the selected proposition.
3	Taps on a proposition on the other side.	
4		Highlights the two selected propositions. A button for verifying a proof appears in the middle of the screen.
5	Taps on the "verification" button.	
6		Verifies the proof and shows the level score.

Alternate Flow: The player verifies an incorrect proof.

	Actor	System
1	Taps on one proposition on one side.	
2		Highlights the selected proposition.
3	Taps on a proposition on the other side.	
4		Highlights the two selected propositions. A button for verifying a proof appears in the middle of the screen.
5	Taps on the "verification" button.	
6		Fails to verify the proof. Gives a score penalty on level clear.

B.2.4.3 Get a hint

Goal: Get a hint in order to find the solution.
Priority: Medium
Actor: User
Description: The user taps on the "Hint" button. The system responds with a hint that could potentially help the user to a solution.

Flow: The player taps the "Hint" button.

	Actor	System
1	Taps on "Hint" button	
2		Displays a hint to the user.

B.2.4.4 Open menu

Goal: Opens the menu while in game.
Priority: Medium

Actor: User
Description: The user taps on the “Menu” button. The system responds by showing a menu, displaying various options.

Flow: The player taps the “Menu” button.

	Actor	System
1	Taps on “Menu” button	
2		Displays a menu to the user.

B.2.4.5 Use menu

Goal: Use the menu to navigate within the program.
Priority: Medium
Actor: User
Description: The user taps a button in the menu which corresponds to what the user is intending to do. The system responds differently, depending on which button in the menu was pressed.

Flow: The player taps the “Main menu” button.

	Actor	System
1	Taps on “Main menu” button	
2		Saves the game and navigates to the main menu.

Flow: The player taps the “Map” button.

	Actor	System
1	Taps on “Map” button	
2		Shows the map of levels.

Flow: The player taps the “Options” button.

	Actor	System
1	Taps on “Options” button	
2		Shows the options.

Flow: The player taps the “Continue” button.

	Actor	System
1	Taps on “Continue” button	
2		Resumes the game.

Flow: The player taps the “Leaderboard” button.

	Actor	System
1	Taps on “Leaderboard” button	
2		Shows the leaderboard.

B.2.4.6 Select level

Goal: Use map to select a level.
Priority: Low
Actor: User
Description: The user taps a level on the map.

Flow: The player taps a level which is unlocked.

	Actor	System
1	Taps on certain level.	
2		Changes to chosen level.

Flow: The player taps a level which is locked.

	Actor	System
1	Taps on certain level.	
2		Informs the actor that the level is locked.

B.3 Class Diagram Model

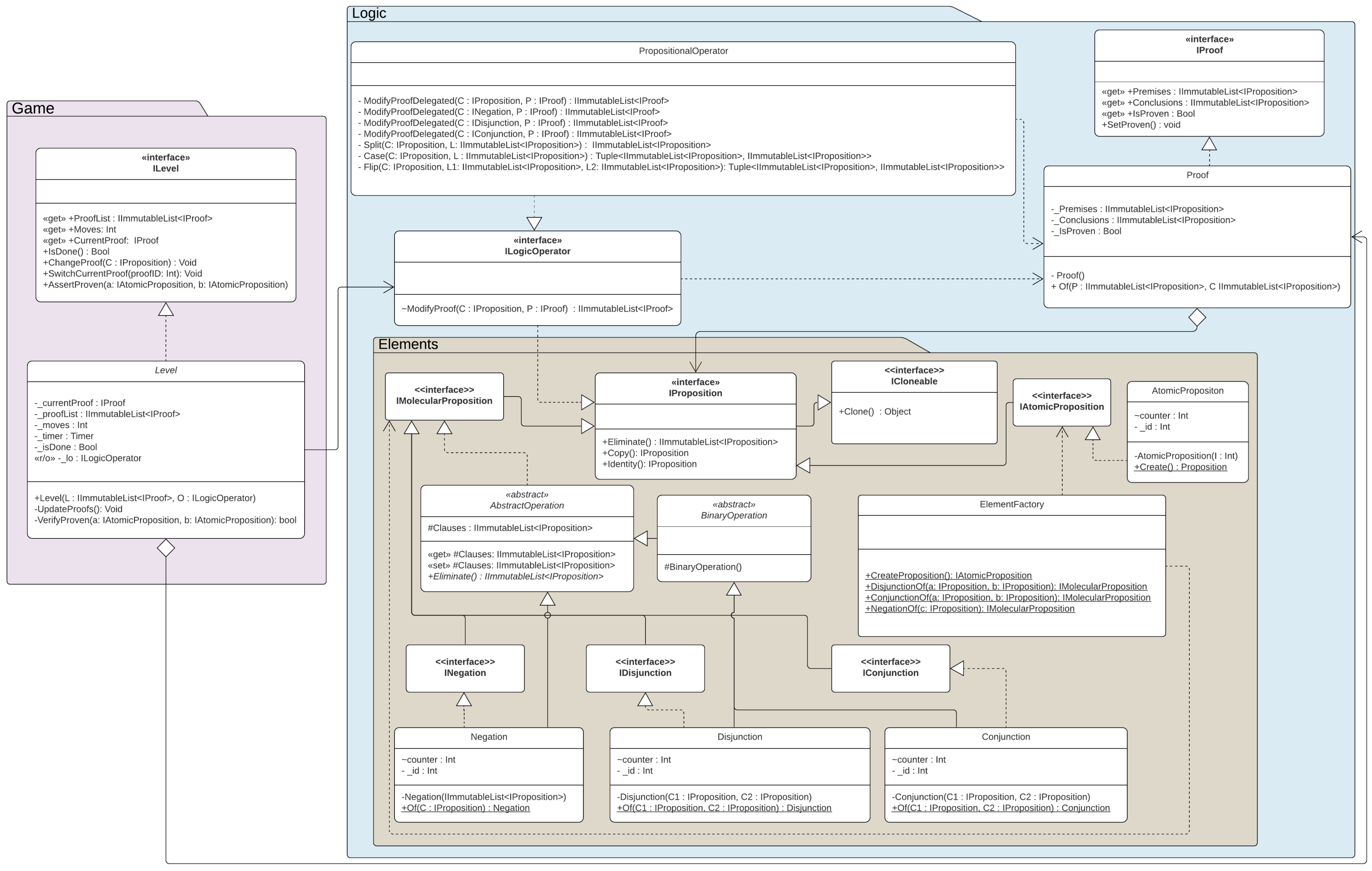

Logic
PropositionalOperator
- ModifyProofDelegated(C : IProposition, P : IProof) : IImmutableList<IProof>
- ModifyProofDelegated(C : INegation, P : IProof) : IImmutableList<IProof>
- ModifyProofDelegated(C : IDisjunction, P : IProof) : IImmutableList<IProof>
- ModifyProofDelegated(C : IConjunction, P : IProof) : IImmutableList<IProof>
- Split(C: IProposition, L: IImmutableList<IProposition>) : IImmutableList<IProposition>
- Case(C: IProposition, L : IImmutableList<IProposition>) : Tuple<IImmutableList<IProposition>, IImmutableList<IProposition>>
- Flip(C: IProposition, L1: IImmutableList<IProposition>, L2: IImmutableList<IProposition>): Tuple<IImmutableList<IProposition>, IImmutableList<IProposition>>
«interface»
IProof
«get» +Premises : IImmutableList<IProposition>
«get» +Conclusions : IImmutableList<IProposition>
«get» +IsProven : Bool
+SetProven() : void
Proof
-_Premises : IImmutableList<IProposition>
-_Conclusions : IImmutableList<IProposition>
-_IsProven : Bool
- Proof()
+ Of(P : IImmutableList<IProposition>, C IImmutableList<IProposition>)
«interface»
ILogicOperator
~ModifyProof(C : IProposition, P : IProof) : IImmutableList<IProof>
Game
«interface»
ILevel
«get» +ProofList : IImmutableList<IProof>
«get» +Moves: Int
«get» +CurrentProof: IProof
+IsDone() : Bool
+ChangeProof(C : IProposition) : Void
+SwitchCurrentProof(proofID: Int): Void
+AssertProven(a: IAtomicProposition, b: IAtomicProposition)
Level
-_currentProof : IProof
-_proofList : IImmutableList<IProof>
-_moves : Int
-_timer : Timer
-_isDone : Bool
«r/o» -_lo : ILogicOperator
+Level(L : IImmutableList<IProof>, O : ILogicOperator)
-UpdateProofs(): Void
-VerifyProven(a: IAtomicProposition, b: IAtomicProposition): bool
Elements
<<interface>>
IMolecularProposition
«interface»
IProposition
+Eliminate() : IImmutableList<IProposition>
+Copy(): IProposition
+Identity(): IProposition
<<interface>>
ICloneable
+Clone() : Object
<<interface>>
IAtomicProposition
AtomicPropositon
~counter : Int
- _id : Int
-AtomicProposition(I : Int)
+Create() : Proposition
«abstract»
AbstractOperation
#Clauses : IImmutableList<IProposition>
«get» #Clauses: IImmutableList<IProposition>
«set» #Clauses: IImmutableList<IProposition>
+Eliminate() : IImmutableList<IProposition>
«abstract»
BinaryOperation
#BinaryOperation()
ElementFactory
+CreateProposition(): IAtomicProposition
+DisjunctionOf(a: IProposition, b: IProposition): IMolecularProposition
+ConjunctionOf(a: IProposition, b: IProposition): IMolecularProposition
+NegationOf(c: IProposition): IMolecularProposition
<<interface>>
INegation
<<interface>>
IDisjunction
<<interface>>
IConjunction
Negation
~counter : Int
- _id : Int
-Negation(IImmutableList<IProposition>)
+Of(C : IProposition) : Negation
Disjunction
~counter : Int
- _id : Int
-Disjunction(C1 : IProposition, C2 : IProposition)
+Of(C1 : IProposition, C2 : IProposition) : Disjunction
Conjunction
~counter : Int
- _id : Int
-Conjunction(C1 : IProposition, C2 : IProposition)
+Of(C1 : IProposition, C2 : IProposition) : Conjunction

Figure B.1: A Class diagram model of the `Core` assembly

B.4 Benchmark Proofs

For testing purposes, a set of benchmark proofs were contructed. They were constructed in three relative difficulty levels, decided and constructed by the group. Solutions to the proofs is not be provided as the purpose for this section is only to display them.

B.4.1 Simple Proofs

This section covers the simple proofs for benchmarking.

B.4.1.1 And Elimination with Modus Ponens

$A \land B, B \to C \vdash C$

B.4.1.2 Implies Elimination

$A, A \lor B \vdash B$

B.4.2 Intermediate Proofs

This section covers the intermediate proofs for benchmarking.

B.4.2.1 Or-condition proof

$(A \lor B) \land C, B \to C \vdash B \lor C$

B.4.2.2 Or-elimination

$A, (A \land C) \lor C \vdash C$

B.4.3 Advanced Proofs

This section covers the advanced proofs for benchmarking.

B.4.3.1 Or-elimination and Or-And conclusion

$(A \lor B) \to C, C \to B, D \lor A, D \land B \vdash (C \lor A) \land B$

B.4.3.2 Conjuction of a disjunction and conjunction

$(A \land \neg B), (D \lor (D \land C)), D \to C,\ B \to (A \lor C) \vdash (A \lor C) \land (D \land C)$

B.4.3.3 Contradiction assumption

$(A \lor B), A \to \neg C \vdash C \to B$