

# A FUNCTIONAL EVENT-DRIVEN FRAMEWORK

A FUNCTIONAL EVENT-DRIVEN FRAMEWORK FOR  
SIMPLIFIED CONCURRENT APPLICATIONS: TECHNICAL AND  
PEDAGOGICAL CONSIDERATIONS

By CHRISTOPHER WILLIAM SCHANKULA, B.Eng.Society

A Thesis Submitted to the School of Graduate Studies in Partial  
Fulfillment of the Requirements for  
the Degree Master of Science — Computer Science

McMaster University © Copyright by Christopher William Schankula,  
December 2024

McMaster University

MASTER OF SCIENCE — COMPUTER SCIENCE (2024)

Hamilton, Ontario, Canada (Computing and Software)

TITLE:                   A Functional Event-Driven Framework for Simplified  
Concurrent Applications: Technical and Pedagogical  
Considerations

AUTHOR:               Christopher William Schankula  
B.Eng.Society (Software Engineering & Society),  
McMaster University, Hamilton, Canada

SUPERVISOR:          Dr. Christopher Anand  
Dr. Spencer Smith

NUMBER OF PAGES:   xxii, 197

# Lay Abstract

Modern software products require graduates experienced with creating interactive programs. To provide this experience, low-floor, high-ceiling activities are needed, ensuring all students can succeed and stay engaged. This thesis investigates creating an Event-Driven Programming (EDP) framework, called TEASync, that allows programmers to create multi-user applications. This thesis describes how the resulting TEASync framework was implemented, and details a study wherein first-year computer science students were given the option of using the framework to create multiplayer games as part of a semester-long design project. Surveys, focus groups, and code compilation data showed that students were able to use the framework to make complex multiplayer games. The data show that the multiplayer option provided experienced students a high-ceiling learning opportunity, while the single-player EDP option maintained a low floor for less experienced students. Those who used the framework reported a lower difficulty level than anticipated and had a measurable increase in engagement in the course.



# Abstract

Modern software products require graduates experienced with creating interactive programs. To properly learn how to build interactive programs, students need projects that present a low barrier to entry (low floor) while retaining a high ceiling to ensure high engagement of more experienced students. In this thesis, we turn to multiplayer game programming using the Event-Driven Programming (EDP) paradigm to fulfill this need. EDP is a type of programming where concurrency is based around the processing of events, which are processed atomically in a central event loop. EDP has been shown to produce more reliable software, precluding many of the common mistakes programmers make with lower-level concurrent programming. We use Elm, which implements EDP in a purely functional setting, processing events as immutable data structure values called messages. This thesis describes the design of a multi-user extension to Elm’s Model-View-Update (MVU) paradigm, called Local-Global Model-View-Update (LG-MVU). We implemented the LG-MVU paradigm in Elm as the TEASync framework. We show how the purely functional language helps to ensure concurrency properties are maintained. We then describe and report on the use of the framework in a first-year computer science course, where students were given the option to use it to implement a video game in a semester-long group project. Surveys, focus groups, and code compilation data were used to ascertain the difficulties and

experiences students had using the framework. The data showed that students were able to use the framework to make complex multiplayer games. Those who used it tended to have more programming experience prior to university, confirming its effectiveness as a high-ceiling activity, while retaining the low-floor option of tool-supported single-player games. Students using TEASync reported a lower difficulty level than initially anticipated, appreciated its simplicity, and had higher levels of engagement in, and enjoyment of, the project and course.

*I dedicate this thesis to my friends and family who have been by my side over the years. I dedicate it especially to my family, who have been my constant inspiration and support — particularly my mom Wendy who’s always there when I need her; my sister Mary who calls me a nerd but secretly thinks I’m cool; my step-father Dave who supports us fully and unwaveringly; my maternal grandmother Jo who does everything for all of us, and late maternal grandfather Ray who supported my love for technology from a young age and instilled in me the importance of giving back to my community; and my paternal grandparents Renate and Frank, whose immigrant eyes taught me to fully love and appreciate my country.*

*Most of all, I dedicate this to my late father Werner, a small engine mechanic who instilled in me a love for tinkering and fixing things by, among a million other things, letting me experiment with making circuits with a 12V car battery in our garage on the shores of Lake Vernon — at the expense of at least one short-circuited and subsequently melted jumper cable.*

*Love you all.*

# Acknowledgements

There are too many people to acknowledge you all by name. I acknowledge all of those who have supported me through the years. First and foremost, I would like to extend my gratitude to my amazing supervisor Dr. Anand and co-supervisor Dr. Smith who have constantly challenged me on my ongoing journey to become a better researcher and educator.

I extend my special thanks to my committee member Dr. Geiskkovitch for her help with the usability study.

I would also like to acknowledge the support, financial and otherwise, of McMaster's Computing & Software Department, and financial support from the government of Ontario and the Natural Sciences and Engineering Council of Canada (NSERC), who made this possible.

I acknowledge the support of all the students in the 1XD3 course who gave of their time to make the usability study a success, filling out surveys and coming to focus groups with a genuine passion to help us improve the tools.

Finally, I am thankful for all the passion of the thousands of K-12 students that have inspired us through our outreach efforts to continue this work, and all the volunteers who have taught lessons, developed tools and curricula, and contributed to the program in so many other ways.

# Table of Contents

<b>Lay Abstract</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>Acknowledgements</b>	<b>vii</b>
<b>Notation, Definitions, and Abbreviations</b>	<b>xviii</b>
<b>Declaration of Academic Achievement</b>	<b>xxiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Goal . . . . .	2
1.3 Contributions . . . . .	2
1.4 Research Questions . . . . .	3
1.5 Research Context . . . . .	3
1.6 Thesis Organization . . . . .	5
<b>2 Related Work</b>	<b>6</b>
2.1 Low Floor, High Ceiling in CS Education . . . . .	6

2.2	Model-Driven Development . . . . .	7
2.3	Requisite Technologies . . . . .	8
2.4	Event-Driven Programming . . . . .	9
2.5	Functional Programming in Education . . . . .	12
2.6	Related Tools . . . . .	13
<b>3</b>	<b>Functional Programming</b>	<b>20</b>
3.1	Functional Programming . . . . .	21
3.2	Hindley-Milner Type Systems . . . . .	26
3.3	The Elm Language . . . . .	29
3.4	Haskell . . . . .	48
<b>4</b>	<b>TEASync Framework Architecture</b>	<b>51</b>
4.1	“Fundamental Theorem of TEASync” . . . . .	52
4.2	Local-Global Model-View-Update (LG-MVU) Architecture . . . . .	53
4.3	Concurrent System Design . . . . .	57
4.4	Alternative Synchronization Schemes . . . . .	61
4.5	Desireable Properties of Global Models and Update Functions . . . . .	73
<b>5</b>	<b>TEASync Framework Implementation</b>	<b>80</b>
5.1	Application Programming Interface . . . . .	80
5.2	Encoder/Decoder (Codec) Generation . . . . .	86
5.3	Module Hierarchy . . . . .	99
5.4	Development Mode . . . . .	102
5.5	Online Collaborative Integrated Development Environment . . . . .	105
5.6	Implementation using Functional Programming . . . . .	108

<b>6</b>	<b>Usability Study Methodology</b>	<b>111</b>
6.1	Overview and Goals . . . . .	111
6.2	Proposition . . . . .	113
6.3	Survey Design . . . . .	113
6.4	Focus Group Design . . . . .	117
6.5	Compilation Statistics Design . . . . .	120
6.6	Threats to Validity . . . . .	122
<b>7</b>	<b>Results</b>	<b>125</b>
7.1	Survey Results . . . . .	125
7.2	Focus Group Results . . . . .	137
7.3	Code Compilation Statistics Results . . . . .	140
7.4	Student TEASync Applications . . . . .	142
<b>8</b>	<b>Conclusions</b>	<b>148</b>
8.1	Summary . . . . .	148
8.2	Research Questions . . . . .	149
8.3	Future Work . . . . .	152
<b>A</b>	<b>Usability Study Instruments</b>	<b>155</b>
A.1	Pre-Implementation Survey Questions . . . . .	155
A.2	Post-Implementation Survey Questions . . . . .	155
A.3	Focus Group Scripted Questions . . . . .	165
<b>B</b>	<b>Code Examples</b>	<b>170</b>

# List of Figures

2.1	STaBL.Rocks code compilation interface . . . . .	16
2.2	STaBL.Rocks project system . . . . .	17
2.3	STaBL.Rocks module history screen . . . . .	19
3.1	Model-view-update dataflow diagram . . . . .	40
3.2	Elm architecture lifecycle . . . . .	41
3.3	Counting example interface . . . . .	46
4.1	Local-global model-view-update dataflow diagram . . . . .	58
4.2	TEASync client sequence diagram . . . . .	59
4.3	Global model-based synchronization lifecycle sequence diagram . . . . .	64
4.4	Global model-based race condition sequence diagram . . . . .	65
4.5	Global model-diff-based race condition sequence diagram . . . . .	67
4.6	Global message-based sequence diagram . . . . .	68
4.7	Global message-based with distributed folding lifecycle sequence diagram . . . . .	71
4.8	Example adventure game state diagram . . . . .	76
5.1	TEASync client module hierarchy . . . . .	101
5.2	TEASync server module hierarchy . . . . .	103
5.3	TEASync development mode Pong example . . . . .	104
5.4	Collaborative TEASync projects on STaBL.Rocks online IDE . . . . .	107



5.5	TEASync server instance overview interface . . . . .	108
5.6	TEASync server instance dashboard . . . . .	109
6.1	CS experience question examples . . . . .	115
6.2	Past game development pre-survey question . . . . .	116
6.3	Pre-survey computer science experience question . . . . .	116
6.4	Post-survey reasons for choosing SP/MP game question . . . . .	118
6.5	Post-survey course and tool experience questions . . . . .	119
7.1	Survey demographics results . . . . .	127
7.2	Prior game programming experience survey results . . . . .	128
7.3	Reported experienced and perceived difficulty of game development aspects . . . . .	129
7.4	Pre-Survey Likert scale question results . . . . .	130
7.5	Choice of single player or multiplayer game results . . . . .	131
7.6	Reported reasons for choosing single vs. multiplayer game . . . . .	132
7.7	Reported experienced difficulty of game development aspects . . . . .	134
7.8	Equal team contribution survey question results . . . . .	134
7.9	State diagram tool Likert statement responses . . . . .	135
7.10	TEASync framework Likert statement responses . . . . .	136
7.11	Reported student happiness with choice of single player vs. multiplayer project . . . . .	136
7.12	Reported project enjoyment . . . . .	137
7.13	Programming experience versus choice of single/multiplayer . . . . .	137
7.14	Results for single player and use of TEASync framework . . . . .	141
7.15	Screens in the <i>Garlic Phone</i> application. . . . .	144

7.16	Screens in the <i>Tap Scotch</i> application. . . . .	146
7.17	Screens in the <i>Brushstroke Journey</i> application. . . . .	147
A.1	Section 1 of the pre-survey, gathering informed consent and demographic data. . . . .	156
A.2	Section 2 of the pre-survey, gathering data about past experience with game development. . . . .	157
A.3	Section 2 of the pre-survey (continued), gathering data about past experience with game development. . . . .	158
A.4	Section 3 of the pre-survey, gathering information about past experiences with game development. (Single player: yes, Multiplayer: no) .	159
A.5	Section 3 of the pre-survey, gathering information about past experiences with game development. (Single player: no, Multiplayer: yes) .	160
A.6	Section 3 of the pre-survey, gathering information about past experiences with game development. (Single player: no, Multiplayer: yes) .	161
A.7	Section 3 of the pre-survey, gathering information about past experiences with game development. (Single player: no, Multiplayer: no) . .	162
A.8	Section 4 of the pre-survey, gathering a baseline of preferences about aspects of computer science. . . . .	163
A.9	Section 1 of the post-survey, gathering informed consent and demographic data, and asking whether they would like to answer the pre-survey questions if they had not already done so. . . . .	164

A.10	Section 2 of the post-survey, asking which type of game the student made. This will determine whether they are asked about their experience making a single player or multiplayer game (Figure A.11 or A.12, respectively).	165
A.11	Section 2 of the post-survey, for students who chose a single-player game.	166
A.12	Section 2 of the post-survey, for students who chose a multiplayer game.	167
A.13	Section 3 of the post-survey, asking agreement with several statements about the course and tools.	168

# List of Tables

4.1	Increment/Decrement Example Sequence . . . . .	62
4.2	Comparison of Synchronization Methods . . . . .	62
4.3	Symmetry Example . . . . .	74
5.1	JSON Format for the <b>TypeDec1</b> Type . . . . .	92
5.2	JSON Format for the <b>BaseType</b> Type . . . . .	92
5.3	JSON Format for the <b>Type</b> Type . . . . .	93
5.4	JSON Format for Select Standard Library Types . . . . .	93
5.5	Binary Format for the <b>TypeDec1</b> Type . . . . .	94
5.6	Binary Format for the <b>BaseType</b> Type . . . . .	95
5.7	Binary Format for the <b>Type</b> Type . . . . .	96
5.8	Binary Format for Select Standard Library Types . . . . .	96
5.9	Example Datatype Encodings and Compression Ratios . . . . .	98
7.1	Shortforms of factors affecting single player/multiplayer choices . . .	133
7.2	Data from STaBL.Rocks IDE [n = 36 users] . . . . .	143
7.3	Selected App Statistics . . . . .	144

# List of Definitions

3.2.1 Top-Level Form of Hindley-Milner Languages . . . . .	26
3.2.2 Types and Type Schemes in Hindley-Milner Languages . . . . .	27
3.2.3 Well-Typedness for Hindley-Milner Programs . . . . .	29
3.3.1 Elm Types . . . . .	35
3.3.2 Labels and Ordered Product Types . . . . .	37
3.3.3 Syntax for Type Signatures . . . . .	37
3.3.4 Syntax for Type and Type Alias Definitions . . . . .	38
3.3.5 The Elm Architecture (Simplified) . . . . .	43
3.3.6 The Elm Architecture (Advanced) . . . . .	47
3.4.1 STM TQueues . . . . .	49
4.2.1 Local-Global Model-View-Update (Simplified) . . . . .	54
4.5.1 Symmetric Global Update . . . . .	74
4.5.2 Associative Global Update . . . . .	75
4.5.3 Idempotent Global Model . . . . .	75
5.1.1 TEASync Simple API . . . . .	81
5.1.2 TEASync Advanced API . . . . .	83
5.2.1 Tokens for Elm Types . . . . .	86
5.2.2 Parser for Elm Types . . . . .	88
5.2.3 Haskell Type for Elm Types . . . . .	89
5.2.4 Find Type Variables Function . . . . .	90

# List of Examples

- 3.3.1 Example Recursive Definition . . . . . 37
- 3.3.2 Simple Counting Program . . . . . 43
- 4.1.1 Fundamental Theorem of TEASync . . . . . 52
- 5.2.1 Example Datatypes . . . . . 97
- B.0.1TEASync Counting Example Code . . . . . 170
- B.0.2Pong Code Example . . . . . 172

# Notation, Definitions, and Abbreviations

## Notation

$A \leq B$	$A$ is less than or equal to $B$
$t \in S$	$t$ is in the set $S$ .
$\mathbf{f} : \mathbf{T}$	$\mathbf{f}$ has type $\mathbf{T}$ — used for Elm types
$\mathbf{f} :: \mathbf{T}$	$\mathbf{f}$ has type $\mathbf{T}$ — used for Haskell types
$\tau$	denotes a monomorphic type
$\sigma$	denotes a polymorphic type
$\tau_1 \rightarrow \tau_2$	denotes a function type
$\lambda x . e$	denotes an anonymous function with the input $x$ and body $e$ . That is, $f = \lambda x . e$ defines the function $f(x) = e$
$\forall \alpha . C$	denotes a polymorphic type with type variable $\alpha$ with a type $C$ that can contain $\alpha$

$\{n : \tau \mid p\}$	denotes the set of items of type $\tau$ that satisfy the Boolean predicate $p$ (values for which $p$ is true)
$\frac{P_1 \dots P_n}{C}$	an inference rule with predicates $P_1$ through $P_n$ and conclusion $C$ .
$\mathbb{N}$	denotes the set of natural numbers, starting from 0
$\mathbb{N}_1$	denotes the set of natural numbers excluding 0. That is, $\mathbb{N}_1 = \{n : \mathbb{N} \mid 1 \leq n\}$
$\mathbb{N}_i^j$	denotes the set of natural numbers from $i$ to $j$ inclusively. That is, $\mathbb{N}_i^j = \{n : \mathbb{N} \mid i \leq n \leq j\}$
$\mathcal{V}$	denotes the set of all symbols
$f \circ g$	denotes function composition
$G_n$	denotes the global model created by processing $n$ global messages
$m_i$	denotes the $i$ th global message

## Definitions

Below is a list of key terms used throughout the thesis:

### Atomic operation

An operation which proceeds from start to finish without being interrupted by another operation. This is a central part of event-driven programming.



<b>Client</b>	A program running on a user's device, either in a web browser or as a standalone application, allowing them to connect to a server
<b>Codec</b>	Short for <i>Coder/Decoder</i> [41]. Codec refers to a scheme used to transmit structured data over a network or store it on a storage device.
<b>Decoder</b>	A function used to turn a structured format (that has been encoded using an encoder) back into a data value in the programming language.
<b>Encoder</b>	A function used to turn a data value of a given type into a structured format suitable for transmission or storage.
<b>Event-Driven Programming (EDP)</b>	
	A type of programming where concurrency is based around the processing of events [13], often atomically.
<b>Fold</b>	Applying a function repeatedly to an initial value of type $b$ and a list of values of type $a$ to reduce the list down to a final value of type $b$ .
<b>Functional Programming (FP)</b>	
	A declarative programming paradigm where a program consists entirely of functions [29, 31].
<b>Map</b>	Applying a function to a list of values of type $a$ to transform the list into another list of type $b$ .
<b>Server</b>	A program running centrally, to which all clients connect and through which they can communicate.

## Abbreviations

<b>AI</b>	Artificial Intelligence
<b>API</b>	Application Programming Interface
<b>BNF</b>	Backus-Naur Form [32]
<b>CS</b>	Computer Science
<b>CS1</b>	Computer Science 1 (First-Year)
<b>EBNF</b>	Extended Backus-Naur Form [32]
<b>EDP</b>	Event-Driven Programming
<b>FP</b>	Functional Programming
<b>GUI</b>	Graphical User Interface
<b>IDE</b>	Integrated Development Environment
<b>IHP</b>	Integrated Haskell Platform framework
<b>JSON</b>	JavaScript Object Notation
<b>LFHC</b>	Low floor, high ceiling
<b>MVU</b>	The Model-View-update Architecture, also known as The Elm Architecture, or TEA
<b>OOP</b>	Object-Oriented Programming
<b>STM</b>	Software Transactional Memory

<b>TEA</b>	The Elm Architecture
<b>TEASync</b>	The Elm Architecture Synchronizer framework

# Declaration of Academic Achievement

I, Christopher William Schankula, declare that the work contained herein is mine. Wherever I have used sources that do not belong to me, I have cited those sources accordingly.

# Chapter 1

## Introduction

Equipping students with the skills needed to work in a 21st-century economy is a growing challenge for educators, both at the K-12 and university levels. Many jurisdictions have taken a while to catch up, e.g. Ontario only formally introducing coding to the mathematics curriculum in 2020 [21]. This contributes to there being a wide range of student backgrounds, including programming abilities and motivations in CS1, which we have anecdotally observed and which has been observed by other researchers [43, 37]. The wide range of backgrounds have been shown to lead to high failure and dropout rates for introductory programming classes [2].

According to Resnick [52], effective educational technology “should provide easy ways for novices to get started (low floors) but also ways for them to work on increasingly sophisticated projects over time (high ceilings)” as explained by Seymour Papert [47]. This concept is called “Low Floor, High Ceiling” (LFHC) [5]. In our context, with no high school computing pre-requisites for CS1, providing a high ceiling also prevents experienced students from becoming bored and disengaged.

For related reasons, many educators have turned to interactive programs in CS1.

To accomplish this, we turn to Event-Driven Programming (EDP), which has become a central concept in modern software development [38]. EDP is a paradigm that focusses on discrete *events* that are processed atomically by an central loop. Languages and frameworks like Elm [10] and React.js have adopted forms of EDP as their primary paradigms. We believe a multi-tiered approach to EDP provides a perfect LFHC classroom environment.

## 1.1 Motivation

The motivation for the thesis is to improve CS1 education, and explore ways to increase student engagement with the material in a heterogeneous classroom containing varied experiences and skill levels with programming and computing in general.

## 1.2 Goal

The goal of this thesis is to explore the use of Event-Driven Programming in a CS1 course and to provide a case study in using EDP to address the problem of a wide range of skill levels in CS1 education. We wish to show that a multi-tiered approach of single and multiplayer games, each using EDP in a purely-functional setting, can provide a low-floor, high-ceiling environment that improves course engagement.

## 1.3 Contributions

This thesis provides three main contributions: 1) the theory and design of a framework and course structure to use event-driven programming, 2) a concrete implementation

of the theory in the form of the TEASync framework, and 3) an evaluation of students' experiences in a class of first-year computer science students. These students were given the option to use TEASync as part of a semester-long design thinking project and their experiences were evaluated using surveys, semi-structured focus groups, and analysis of data from the online Integrated Development Environment (IDE) used by students.

## 1.4 Research Questions

The following are research questions under study in this thesis. They will be referenced throughout the thesis by referring to their RQ number.

RQ1. How can EDP in a functional context be extended to support multi-user applications?

RQ2. What are the measurable differences in course engagement between the single player and multiplayer groups?

RQ3. What evidence is there that this approach successfully provides a LFHC environment for students?

## 1.5 Research Context

This section provides more context in which the research has taken place, including information about McMaster University, as well as the McMaster Start Coding and STaBL Foundation organizations.

### 1.5.1 McMaster Start Coding

McMaster Start Coding has been teaching K-12 and university students computer science concepts for over 15 years. The mission of the club and accompanying research group is to give new programmers the tools and confidence they need to pursue STEM careers. Thanks to undergraduate and graduate student volunteers, in the past 8 years, McMaster Start Coding has delivered free coding sessions to over 30,000 students. One area of focus is on underprivileged and underrepresented youth in the Hamilton area. Since the COVID-19 pandemic, virtual instruction has allowed the organization to reach beyond our geographical region into other parts of Canada and around the world.

### 1.5.2 STaBL Foundation

Founded in 2022, the registered Canadian charity Fondation STaBL Foundation's<sup>1</sup> mission is to develop and provide scalable team-based learning tools to teach the next billion programmers around the world. The Foundation works closely with McMaster Start Coding to partner with community organizations in Ontario, such as Lumenus Community Services<sup>2</sup> and school boards to provide tailored programs to their particular needs. STaBL is also partnering with organizations in countries like India, Nigeria, St. Lucia, Guatemala, and Burundi to provide international outreach to those in developing countries.

---

<sup>1</sup><http://stablfoundation.org>

<sup>2</sup>Lumenus Community Services “offer[s] a broad range of high quality mental health, developmental, autism and early years intervention services to children, youth, families, and individuals across Toronto.” (<https://www.lumenus.ca/>)



## 1.6 Thesis Organization

This thesis is divided into a total of 8 chapters, including this introduction:

- Chapter 2 details motivational past work in the area of event-driven programming and model-driven development.
- Chapter 3 provides an outline of functional programming practices, Hindley-Milner type systems, the Elm language, model-driven design, as well as requisite technologies and related tools. The information in this chapter is likely nothing new for experienced functional programming practitioners, but is included for those less experienced with FP.
- Chapter 4 describes the theory and architecture of the TEASync framework, from a high-level mathematical and software architectural point of view.
- Chapter 5 delves into more details about the implementation of the framework.
- Chapter 6 describes the methodology used for the design and results analysis of the usability study.
- Chapter 7 provides and analyzes the results of the usability study and shows some example applications created by users of the framework.
- Chapter 8 provides conclusions, recommendations, and details of future work in this area.
- The Appendix provides raw data and instruments such as survey questions.

# Chapter 2

## Related Work

This chapter discusses prior work and research in the area. The purpose of this chapter is to provide the necessary background to motivate the rest of the thesis, including background research on Low Floor, High Ceiling (LFHC) teaching, Model-Driven Development (MDD), requisite technologies used to create the framework and their properties, the use of Event-Driven Programming (EDP) and Functional Programming (FP) in education, and some related work developed in our lab.

### 2.1 Low Floor, High Ceiling in CS Education

The LFHC principle was first introduced in the 1970s by Seymour Papert [5, 47]. He coined the term while describing his programming language Logo, which is considered the first language meant to be accessible to children while being useful for adults [5, 47].

Maiorana [39] describes a low floor as being an activity (or, more generally, a curriculum) that provides an entry point suitable to all learners. That is, the activity

or some subset of the activity should be accessible to all learners. However, at the same time, the activity must provide a high ceiling: one which “support[s] the curiosity of all learners” [39]. In other words, the fact that the activity is accessible to all learners should not preclude those with the ability and interest to continue to explore and expand upon their learning.

The LFHC design principle has been adopted by many educators and researchers, including in the design of Scratch [66], ScratchJr [5, 57], a web-based Python IDE [4], and as a guiding principle for CS curricula, including those aimed at K–2 [68], grade 4–6 (age 9–11) [26] and K–12 [36] students and teachers thereof.

This work will investigate the role of event-driven programming with a single player and multiplayer option in creating a classroom environment that provides a low floor and high ceiling for students.

## 2.2 Model-Driven Development

Model-driven development (MDD) is a software engineering methodology that applies

“models and model technologies to raise the level of abstraction at which developers create and evolve software, with the goal of both simplifying (making easier) and formalizing (standardizing, so that automation is possible) the various activities and tasks that comprise the software life cycle” [24].

Thus, MDD aims to capture the details of the software engineering process in a single source of truth, and generate artifacts (code, documentation, etc) from this.

TEASync uses concepts from MDD in that the types created by the user are used to generate the encoders and decoders needed for client-server communication. This approach might best be described as “data-structure-driven” because the data structures play the role of the model in MDD. The system parses the data structures and generates server code and a shell that wraps around the user’s client code and handles all communication and synchronization.

## 2.3 Requisite Technologies

This section lists some useful technologies that were used in the creation of the framework, including their useful properties.

### 2.3.1 WebSockets

The WebSockets protocol allows for two-way communication between a client and a server [20]. This was created to replace older methods such as long-polling which abused HTTP, leaving a connection open until the server had data to send back, and using separate connections for the client to send data to the server [20], which has the downsides of high overhead and additional unnecessary open HTTP connections.

After an initial handshake, the client and server can send data to each other at will using “frames”, which can be referred to as “messages”. Furthermore, WebSockets are layered on TCP [20], which ensures the ordering of messages received by the client/server is the same order as they are received by the server/client, respectively. This is a useful property for the TEASync framework, as will be explained in Chapter 4.

### 2.3.2 JavaScript Object Notation

JavaScript Object Notation (JSON) is a data format that is made to mimic data types in the JavaScript language [51]. The full specification of JSON syntax is omitted for brevity and because only a subset of JSON is necessary for TEASync. It can be viewed in Pezoa et al. [51].

Since Elm compiles down to a JavaScript file when compiled, it has inbuilt support for JSON objects, and was therefore a natural choice for human-readable communication amongst clients. Since Elm is a strongly-typed language, it is necessary to formally supply encoders and decoders for converting data types to JSON and vice versa. This is different from what JavaScript programmers are used to, where JSON objects can be serialized and deserialized with general-purpose functions. Chapter 5 describes precisely how the TEASync framework generates encoders and decoders from the programmer’s data types, as well as the JSON objects they produce.

## 2.4 Event-Driven Programming

EDP is a type of programming where concurrency is based around the processing of events [13]. This is instead of the use of IO in threads with blocking. In the case of threads, concurrency is achieved by blocking (waiting) in a thread and resuming the processing of another thread. According to Dabek et al. [13], the main advantage of threads is that it allows the overlapping of IO and computation while preserving what appears (to the programmer) to be a serial programming model. However, this strength is also its biggest weakness, as it introduces concurrent execution where it is not needed [13]. The programmer must thus explicitly write the logic to achieve

thread safety, and must be careful to protect shared data using locks to avoid multiple threads trying to access it at once [13]. Since it is left up to the programmers to get this concurrent logic right, this leads to robustness issues in software [13].

Event-driven programs, on the other hand, work by processing *events*, which are, for example, the arrival of a piece of information, the result of a user interacting with the program by clicking a button or typing on their keyboard. They work by registering a *callback*, or a function to be executed when an event finally happens. Typically, event-driven programs are structured around a central loop that waits for events and processes them atomically, also known as indivisibly: once an event begins processing, it continues until it is fully processed. Several authors show that the EDP paradigm leads to more reliable software [13, 15].

A 2021 survey of EDP for education was published by Lukkarinen et al. [38]. Despite its noted benefits, new programmers face challenges while learning EDP. Much of the difficulty comes from reasoning about their program’s behaviour. Since execution order depends significantly on the order of events, which Woodworth and Dann [67] calls an *inversion of control*, EDP is significantly different from traditional sequential programming.

Many studies have been published about teaching EDP to students over the decades, with Woodworth and Dann themselves first teaching EDP by starting with console-based programs and then moving onto GUI-based programs. A few years later, Christensen and Caspersen [9] published a framework called the *Presenter* framework as another approach to teaching EDP using Object-Oriented Programming (OOP).

Since then, EDP has been used to teach several different topics in CS1, including

image manipulation [64], graphics [1, 23, 65], design patterns [50], and games [22, 17, 19].

Clearly, the effects of EDP on education and practice has been an active area of research since the 1990s [38]. Despite this, Lukkarinen et al. [38] concluded that research in this area remains underdeveloped:

While most studies focus on bachelor’s level education in universities, there has been substantial work in K-12 level, as well. Few courses address EDP as their main content—rather it is most often integrated with CS1, CS2, or computer graphics courses [...] Moreover, very little of deliberate experimental scientific research has been carried out to explicitly address teaching and learning EDP. Consequently, while so-called experience reports, tool papers, and anecdotal evidence have been published, this theme offers a wide arena for empirical research in the future [38].

In fact, of the 105 papers that were analyzed in this study [38], only 52 of them provided any quantitative results. The paper goes on to suggest several areas of future research. The three predominant languages in their review were Java, App Inventor, and Scratch. OOP was a common focus, with functional languages rarely discussed. The word “functional” does not even appear in the titles of any of the papers in their citation list.

Lukkarinen et al. goes on to pose several research questions for future focus. Two questions of particular interest in the current work are:

- “*Programming Languages*. How does teaching and learning EDP happen with different programming languages? What differences

there are when learning EDP with Java, Python, C++, or some other languages? Obviously, block-based languages have a different learning process. Can students transfer what they have learned, for instance, in App Inventor or Scratch into implementing EDP using text-based languages?” [38]

- “*Functional Languages*. How does learning EDP differ, if students use a functional language for building interactive applications?” [38]

## 2.5 Functional Programming in Education

This section discusses the role of Functional Programming (FP) in education. The principles of functional programming are introduced in Chapter 3.

Functional programming has been taught in first-year computer science since at least 1993, when Joosten et al. [34] discussed its benefits in introducing algorithmic thinking without unnecessary and distracting syntactic overhead. Their results in comparing to an imperative language found that students using functional programming to solve the given problem introduced over 3x more functions and had a greatly increased coverage of the design problem than the control group of imperative programmers [34].

Other researchers have noted FP’s closer alignment to mathematical thinking, minimal side effects, and fewer lines of code for many tasks as being advantageous, especially for writing AI code [59]. Chakravarty and Keller [8] argue that purely functional programming—that is, functional programming in languages with immutability and pure functions, see Section 3.3.2—have advantages for teaching concepts to first-year classes, but only if the class focuses on concepts rather than FP itself.



It is for related reasons we have used FP to successfully introduce algebraic thinking [14] to K-PhD students, and have observed that it increases social cohesion and mathematics knowledge [69]. We have also found state diagrams to be a successful abstraction when teaching interactive programming using FP [48] (see Section 2.6.2).

## 2.6 Related Tools

The following is a brief summary of other tools that have been developed and support the TEASync framework in this study.

### 2.6.1 GraphicSVG

The GraphicSVG framework [69] is a vector graphics framework developed to support teaching algebraic thinking. Algebraic thinking is characterized as the cognitive skills needed to promote algebra [35], the ability to detect patterns and put different pieces together in a constrained way. Algebra has been shown to be a major gateway to high school success [55].

The framework allows students to make graphics and animations declaratively, which, especially when embedded in a functional programming language like Elm, matches students' intuition from their math courses [69].

TEASync was designed to allow for multiple rendering frontends, but for purposes of the course, students used their GraphicSVG knowledge learned in the previous semester to create their games.

### 2.6.2 State Diagram Tool

The State Diagram tool [48] is an MDD tool developed to allow students to visualize Elm Model-View-Update (MVU) applications. The tool allows students to describe an application or game as a state diagram, which is then used to generate initial Elm MVU code, including the `init`, `Model`, `update` and a basic `view` function which can be subsequently expanded upon in code. The tool has been used to teach students from grade 4 up to undergraduate and graduate students. We have shown [48] that even young students were able to interpret and understand the state diagram model of computation and create complex adventure games and other applications.

### 2.6.3 STaBL.Rocks Online Integrated Development Environment

The STaBL.Rocks online Integrated Development Environment (IDE) is a system developed using the Integrated Haskell Platform (IHP) [16]. On this system, developers can create, share, get help with, and collaborate on Elm modules and projects. It is used by thousands of elementary-to-PhD students each year. This subsection will detail this system, as concepts like modules, activity types, and the project system will be relevant later in our discussion of the TEASync framework.

#### Overview

The system is broadly broken up into several components:

- Elm Modules Screen
- Code Compilation Screen

- Project System
- Mentor System
- Admin System

### **Elm Modules Screen**

The Elm Modules screen lists all the modules that the programmer has created. It allows them to create modules from pre-made activity types. Activities are templates that have defined starting code, hidden code, library imports, and associated documentation. Students have a list of activity types they are allowed to access, which can be controlled by administrators. For instance, student developers in a certain class may have special access to activity types which others do not have access to.

### **Coding Screen**

The code compilation screen (see Figure 2.1) is where students can write and test their modules. The left-hand side of the screen (Figure 2.1 (A)) allows students to code using an online code editor, with the output on the right (Figure 2.1 (C)). Upon clicking the blue compile button (Figure 2.1 (B)), the code is sent to the server to be compiled and the resulting output is shown on the right. Figure 2.1 (D) shows the mentor chat, where developers can get help with their code from volunteer mentors, and a resource bar, Figure 2.1 (E), shows helpful links to resources like documentation and online YouTube-based help videos to help developers.

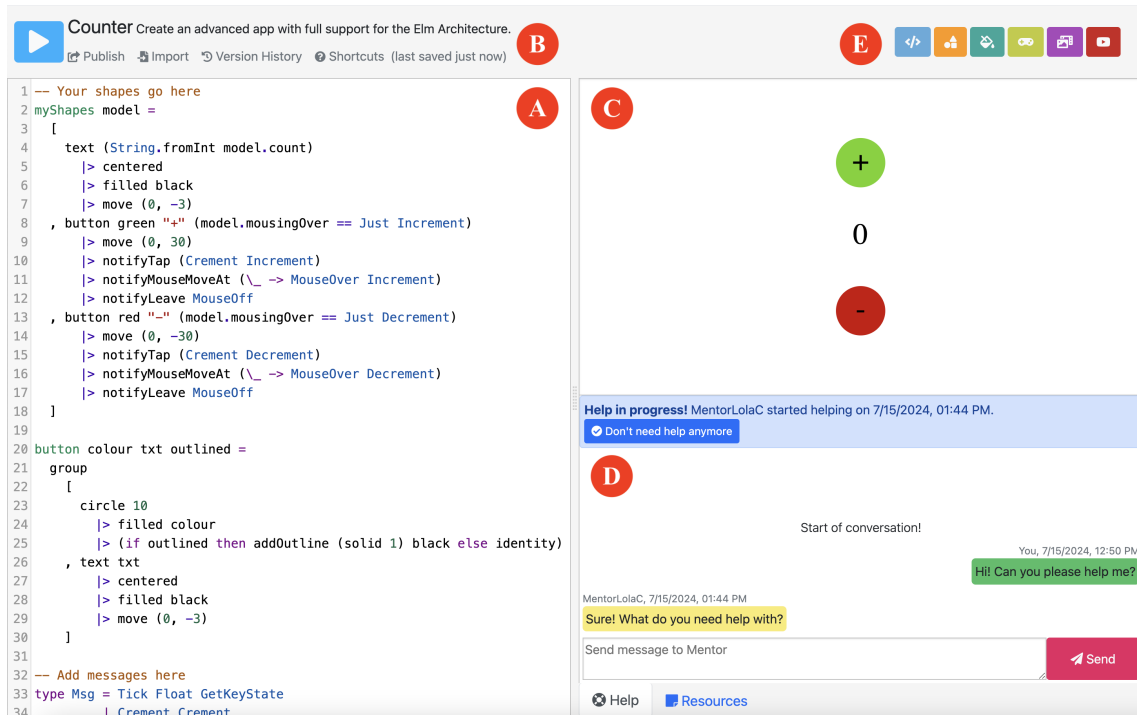


Figure 2.1: The STaBL.Rocks code compilation interface. In (A), developers write their code directly in the browser-based text editor. (B) provides actions like the blue compilation button, the ability to publish and share their code/output, the ability to access the module’s version history, and get help with keyboard shortcuts. (C) is the output box where the compiled program is displayed. (D) provides a mentor help chat for developers to get help with their code. Finally, (E) has links to several helpful resources such as a the interactive Shape Creator, YouTube help videos, and package documentation.

**BorgarWord**

All ▾

Enter Module Name

Filter

Clear

View Who Has Access

+ New Module

+ New Folder









Name	Last modified
 <b>Ant</b>	5 days ago
 <b>ATUTORIAL</b>	2 days ago
 <b>ATUTORIALw</b>	2 days ago
 <b>Bee</b>	5 days ago
 <b>Beetle</b>	5 days ago
 <b>Butterfly</b>	5 days ago
 <b>Choose4</b>	2 days ago
 <b>CustomCafe</b>	1 day ago

Figure 2.2: STaBL.Rocks allows developers to create projects which can be shared with others. Modules without errors can be released along with a release message, and doing so allows other modules to import those modules.

## Project System





















The STaBL.Rocks project system (see Figure 2.2) allows module-level collaboration with a simple version control system. Developers can add collaborators to their project, and can create an arbitrary number of modules to their project.

Modules inside a project have the extra option to be “released”. Releasing a module can only be done when the module compiles successfully. A release is accompanied by a note from the developer about changes since the last release. The released version of the module is the version that will be imported into other modules. This allows developers to have a safe module system where changes made by one developer do not affect other developers. While not as powerful as version control systems like Git, this system is designed to be simple for new developers and prevent importing an errored module into another. Figure 2.3 shows the version history screen, with releases listed in the second-last column.

### 2.6.4 Petri App Land Framework

The Petri App Land (PAL) framework was a precursor to TEASync, and was successfully used to build applications in a collaborative environment with grade 5-12 students, including a game for learning math concepts and an application for newcomer Canadian youth [53]. PAL was based on the concept of Petri Nets, a model for concurrency. PAL included tools for model-based client and server code generation, but required developers to write further code on both the server and client. Ultimately, its complexity proved difficult for new developers and much mentor intervention was required. TEASync’s approach of only editing code in the client was inspired by PAL’s shortcomings.

Module History
×

Author	Version	Date	Compiles	Released	Actions
Tranm72	6	1 day ago 7/13/2024, 10:54 PM	✓		  
ProfAnand	5	2 days ago 7/13/2024, 04:55 AM	✓		  
ProfAnand	4	2 days ago 7/13/2024, 04:54 AM	✗		  
ProfAnand	3	2 days ago 7/13/2024, 04:54 AM	✗		  
ProfAnand	2	2 days ago 7/13/2024, 04:29 AM	✓		  
ProfAnand	1	2 days ago 7/13/2024, 04:21 AM	✓		  

Show All
Close

Figure 2.3: The module version history screen for a module inside a project. Multiple people can edit the module at different times, and a new column indicates if a version is a released version of the module. The latest release of the module (in this case, version #2) is used as the version to be imported into other modules, providing a safe way of handling module hierarchy.

## Chapter 3

# Functional Programming

This chapter provides background information necessary to understand the context and concepts that are needed throughout the thesis. Practitioners of FP can likely skim or skip this chapter outright as many of the concepts should be familiar. They are included here for informational purposes.

Firstly, in Section 3.1, we introduce functional programming as well as the common features and properties of functional languages, many of which support the creation and desirable properties of the TEASync framework. This is followed in Section 3.2 by an overview, notational description of, and formal description of Hindley-Milner (H-M) type systems, concepts upon which the TEASync framework bases its data model and codec generation. Next, we introduce the Elm language itself in Section 3.3, including its history, desirable properties, its H-M type system, and Elm Architecture. We then briefly introduce the history of the Haskell language and some useful libraries and frameworks in Section 3.4.



## 3.1 Functional Programming

Most programming languages that are popular<sup>1</sup> today are of the *imperative* style [60]. The model of computation in these languages is to have an implicit state, which is modified by a series of side-effects brought on by commands [29]. For instance, consider the problem of summing the numbers from 1 to 10. To do so in the imperative Python language, we can use the following code:

```
result = 0
n = 10
while n > 0:
    result = result + i
    n = n - 1
```

In this example, we are using assignment statements to modify the state of the variables. After we run this program, the value will be stored in the `result` variable. The implicit state is the value of all the variables in the program.

On the other hand, *declarative programming* is a type of programming where there is no implicit state [29]. The program contains expressions or terms to be evaluated, rather than a sequence of commands to be executed.

Functional programming is a declarative programming paradigm where a program consists entirely of functions [29, 31]. The main program (also known as the entry point to the program), is usually written in a function called `main` which itself calls other functions in the program. Functions in these languages are more like ordinary mathematical functions than most procedural languages, in that they are often pure

---

<sup>1</sup>Of the top 10 in this Statista data from 2023, the only one that is considered declarative is SQL.

functions without side-effects [31].

Let’s revisit the example from before. In Elm, the code would be expressed as a function:

```
sumN n =  
    if n == 1 then  
        1  
    else  
        n + sumN (n - 1)
```

Notice in this example that we keep the state of the program explicit; that is, the parameter `n` carries around the state *explicitly* rather than implicitly [29]. Note as well that the `if` construct is an *expression* and not a control flow command [29]. If one wanted to compute the sum from 1 to 10, one would simply have to call the function using 10 as the parameter, i.e. `sumN 10`. The *value* of the expression `sumN 10` when evaluated would be equal to 55.

### 3.1.1 Common Features and Properties of Functional Languages

In a 1989 paper “Why Functional Programming Matters” [31], Hughes describes functional programming’s ability to “push back [...] the conceptual limits on the way problems can be modularized” [31]. Functional programming’s ability to increase modularization, Hughes says, is “glued together” by higher-order functions and lazy evaluation.

## Higher-Order Functions

Higher-order functions are functions that take other functions as input. For example, common higher-order functions in Elm are `List.map` and `List.foldl`<sup>2</sup>:

```
map      : (a -> b) -> List a -> List b
foldl    : (a -> b -> b) -> b -> List a -> b
```

As any seasoned functional programmer knows, the `map` function takes a function and applies it to a list of values, producing a new list of values, and the `foldl` function takes as input a “reduction” function, a starting value and a list of elements, and applies the given function on every element of the list, “folding” or “reducing” the list down to a given type of value `b`.

Hughes argues that this first-class support for functions, allowing them to be passed in as arguments to other functions, is a powerful form of “glue” allowing for increased modularity. For instance, he argues, it is simple to create a `sum` function to sum up a list of numbers `ns`:

```
add x y = x + y
sum ns = List.foldl add 0 ns
```

## Lazy vs. Strict Evaluation

The second form of glue Hughes argues for is known as lazy evaluation. While Elm is not a lazily-evaluated language, languages like Haskell support lazy evaluation. Hughes gives the example of function composition to discuss why this is an important feature. For instance, consider the Haskell function composition operator<sup>3</sup>:

---

<sup>2</sup><https://package.elm-lang.org/packages/elm/core/latest/List>

<sup>3</sup><https://hackage.haskell.org/package/base-4.19.1.0/docs/Prelude.html#v:>

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

This function takes two functions, e.g. `f` and `g` and *composes* them together, such that  $(f \cdot g) x = f (g x)$ . In a lazily-evaluated language like Haskell, the result of  $(g x)$  need not be stored explicitly and instead the computations of `f` and `g` can be interwoven [31]. This means that computations that wouldn't be otherwise possible to fit into memory become possible, which was particularly necessary for the era in which the paper was written.

Under strict evaluation, on the other hand, intermediate values are always evaluated down to their “most evaluated” form; there are no intermediate, unevaluated results. Elm is one such language.

## Referential Transparency

Referential transparency, which is related to purity [28], is another important characteristic of functional programming. While in the Python example, the `=` symbol is a *destructive update* of the variable on the left-hand side, assigning a new value to the variable, in functional languages `=` means actual equality [28]. For instance, `sumN 3 = 6` in the true sense of mathematical equality. This matches programmers' intuitions from their mathematics courses, and anecdotally we have observed that this is more natural for beginner programmers.

### 3.1.2 Advantages

In a follow-up to his original 1989 paper, called “Why Functional Programming Mattered”, Hu et al. (including Hughes himself) discusses how Hughes' vision for functional programming from the original paper has since gone on to become one of the

most highly-cited in the field and more accepted in general software development. In particular, higher-order functions and lazy evaluation have been shown to add crucial advantages in the software development process. The authors go on to cite evidence including the fact that many languages like Python and Java have added lambda functions and other functional features, and the prevalence of functional programming conferences as evidence of functional programming’s rise in popularity.

### 3.1.3 Drawbacks

While functional programming has strong foundations and draws on a wealth of mathematical knowledge, there are some potential drawbacks to some of these properties, both technically and pedagogically. One potential risk for functional languages’ higher level of abstraction is the potential for students to “learn the patterns” rather than learning the underlying concepts [61]. In our experience, we have found that students who already know imperative-style languages like Python are more likely to be resistant to learning functional programming. This is likewise observed by [30] in his book *The Haskell School of Expression*, who states “There will be a tendency to rely on old habits when writing new programs” but that those who can resist those tendencies “find that many of the things that they learn about functional programming can be applied to imperative and object-oriented languages. . .”. This is an approach we preach to our students, but from our experience, Elm seems to be more quickly embraced by the grade 4 through 12 students we visit than by the first year students we teach.

In practice, lazy evaluation is a mixed bag in terms of its benefits. Some studies have found that lazy versions of algorithms used increased computational power

the majority of the time, leading to higher energy usage from both the CPU package and DRAM [40]. Anecdotally, lazy evaluation is sometimes a hindrance to new programmers, especially when it comes to odd memory management behaviour and computations “blowing up” from a memory and CPU point of view due to unevaluated thunks [6]. For this reason, Elm was designed early on as a strict language, one where all values are evaluated right away [10]. Furthermore, lazy evaluation has been shown to even be a security risk in certain circumstances [7].

## 3.2 Hindley-Milner Type Systems

Hindley-Milner (HM) type systems were first described by Hindley in a 1969 paper [27], and later reiterated by Milner in 1978 [42]. Although the semantics for type inference are beyond the scope of what is needed for this thesis, the basic structure is important to understand the encoder/decoder generation in the TEASync framework. This section will elaborate the general definition of HM systems, and Section 3.3.4 will give a definition specific to the Elm language. In his 2000 thesis [58], Sulzmann describes the syntax and semantics of Hindley-Milner languages. Much of the rest of this section draws upon that thesis.

Hindley-Milner languages contain the following top-level form [58] shown in Definition 3.2.1.

### Definition 3.2.1: Top-Level Form of Hindley-Milner Languages

$$\begin{aligned} \text{Values } v &::= x \mid \lambda x . e \\ \text{Expressions } e &::= v \mid e e \mid \text{let } x = e \text{ in } e \end{aligned} \tag{3.2.1}$$

The different types of values are variables (represented by  $x$ ) and lambda-abstraction.

The different types of expressions are values themselves, function application (where an input parameter is applied to a function to produce a value) and finally, let expressions which are used to bind variables.

Further, the types of these values and expressions (adapted from [58]) are defined in Definition 3.2.2.

**Definition 3.2.2: Types and Type Schemes in Hindley-Milner****Languages**

$$\mathbf{Types} \ \tau ::= \alpha \mid \tau \rightarrow \tau \tag{3.2.2}$$

$$\mathbf{Type Schemes} \ \sigma ::= \tau \mid \forall \alpha. C \Rightarrow \sigma$$

where  $\alpha$  represents a type variable which may or may not appear in the type scheme  $\sigma$

Hindley-Milner types are divided into two categories: Types, or *monomorphic* types, and Type Schemes or *polymorphic* types. Monomorphic types include types like `String`, `Int`, and `Bool`. Polymorphic types are types with type variables, where the type variables are monomorphic types [58]. By convention, we follow Sulzmann’s convention of using  $\tau$  to range over monomorphic types and  $\sigma$  to range over polymorphic types. Note that as a consequence of the definition, we cannot have polymorphic types as the input or return type of a function. This is known as Higher-Rank polymorphism [18], which is supported by language extensions in Haskell, but not in Elm, and so it is not considered here.

### 3.2.1 Well-Typed Program Inference Rules

A type system uses *inference rules* to define acceptable types in the language [49]. These inference rules are of the following form:

$$\frac{P_1 \dots P_n}{C} \text{ NAME} \quad (3.2.3)$$

where  $P_1$  through  $P_n$  are the premises and  $C$  is its conclusion [49]. These rules can be viewed as an implication with conjunction between the premises, i.e.  $P_1 \wedge \dots \wedge P_n \Rightarrow C$ , but which also can be seen to have an implicit  $\forall$  quantifier for the seemingly-unbound variables in the rule. The  $\forall$  is usually left implicit for the sake of conciseness. Furthermore, each rule is accompanied by a NAME on the right-hand side.

These can be understood in two ways [49]:

- “If all of  $P_1$  through  $P_n$  hold then  $C$  holds.”
- “To prove  $C$  we must prove  $P_1$  through  $P_n$ .”

Definition 3.2.3 shows the type judgment rules for Hindley-Milner programs, as defined by Sulzmann [58]. A typing judgment is written as  $\Gamma \vdash e : \sigma$ , where  $\Gamma$  is a set containing free variables and their types,  $e$  is an expression in the language, and  $\sigma$  is a type scheme. A typing judgment is said to be *valid* if it is derivable from the typing rules [58]. The symbol  $\Gamma_x$  means the type environment not containing the variable  $x$ , i.e.  $\{y : \sigma \mid y : \sigma \in \Gamma . y \neq x\}$  [58]. And the syntax  $\Gamma_x . x : \sigma$  means extension of a type environment, i.e. inserting  $x$  into the set [58]. We must add  $x$  into the environment when introducing new variables whether through the LET or ABS (function abstraction) rules.  $fv(\Gamma)$  is a function representing the set of free variables in  $\Gamma$ . Finally,  $[\bar{\tau}/\bar{\alpha}]$  represents substituting the sequence of type variables  $\bar{\alpha}$



with the sequence  $\bar{\tau}$ . This is used to *eliminate* type variables in a quantification by substituting types in their place.

### Definition 3.2.3: Well-Typedness for Hindley-Milner Programs

The following type judgments describe properly-typed Hindley-Milner programs, as described by Sulzmann [58]:

$$\Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma) \text{ VAR} \tag{3.2.4}$$

$$\frac{\Gamma_x . x : \tau \vdash e : \tau'}{\Gamma_x \vdash \lambda x. e : \tau \rightarrow \tau'} \text{ ABS} \tag{3.2.5}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2} \text{ APP} \tag{3.2.6}$$

$$\frac{\Gamma_x \vdash e : \sigma \quad \Gamma_x . x : \sigma \vdash e' : \tau'}{\Gamma_x \vdash \text{let } x = e \text{ in } e' : \tau'} \text{ LET} \tag{3.2.7}$$

$$\frac{\Gamma \vdash e : \tau \quad \bar{\alpha} \notin fv(\Gamma)}{\Gamma \vdash e : \forall \bar{\alpha}. \tau} \forall \text{ INTRO} \tag{3.2.8}$$

$$\frac{\Gamma \vdash e : \forall \bar{\alpha}. \tau'}{\Gamma \vdash e : [\bar{\tau}/\bar{\alpha}] \tau'} \forall \text{ ELIM} \tag{3.2.9}$$

where the overbar  $\bar{x}$  indicates that  $x$  is a vector of variables

## 3.3 The Elm Language

For the last several years, we have used the Elm language to teach students right from grade 1 through PhD, and have found success in using it to teach the critical skills of algebraic thinking [14], which is critical for success in high school and post-secondary education, as well as teamwork and social cohesion [69]. This section introduces Elm's

history, its favourable properties for the current work, and its architecture, which will be relevant in Chapter 4.

### 3.3.1 History

Elm is a functional programming language created by Harvard student Evan Czaplicki in 2012 [10]. The original architecture for the language was called concurrent functional reactive programming (FRP), which is a general framework for “programming hybrid systems in a high-level, declarative manner” [62]. Programmers in Elm programmed using *Signals*, which were queues into which messages (events) could be inserted by one part of the program and processed by another. FRP was later replaced by a simplified architecture known as The Elm Architecture (TEA) in 2016, which can be seen as a single-signal (queue) FRP system. This change was brought on by the fact that most Elm programmers at the time used a single FRP signal to emulate what became the Elm Architecture anyway, and thus signals were removed and fully replaced by the Elm Architecture [11]. Section 3.3.5 discusses TEA in much more detail.

### 3.3.2 Purely Functional: Immutability and Pureness

Two key properties of Elm are its *immutability* and *pureness*. Together, these properties make what is known as a *purely functional* language, of which Elm is one [10].

Immutability means that all values in the language are said to be immutable: they cannot be changed once they are instantiated. Pureness, on the other hand, is a guarantee that a given function will always return the same output when given a specific input [10]. According to Czaplicki [10], these two properties are rare for

modern languages despite their benefits for program reliability and maintainability.

The pureness property in particular is an important property for the current work, as will be explored in Chapter 4. Anecdotally, we have found Elm’s pureness to be a strength for new programmers who already have this intuition from their mathematics classes. Furthermore, this allows us to support their existing mathematical instruction well. We have found great success teaching grade 4 through 12 students about functions using a visual approach, where inputs are colours and outputs are, for instance, a flower with petals of that colour [14].

### 3.3.3 Totality

Another one of Elm’s important properties is its *totality* guarantee. Totality ensures that all cases in an Elm program must be covered; that is, Elm programs are guaranteed not to generate common runtime exceptions or crashes as in other languages<sup>4</sup>

Consider the following code snippet in Python:

```
def specialNumber(n):  
    if n == 42:  
        return "The answer to life, the universe, and everything"
```

This function is obviously *partial*; what if we give it 41, or -53 or 53.5 billion? The function is undefined for those values; it will simply return nothing. In Python, this value is called *None*. While this is an extreme example, this is a problem because it makes it possible for this code to run in production for many months or even years

---

<sup>4</sup>Division by zero causing NaN values are still possible, which can cause undefined behaviour. Furthermore, infinite recursion can cause an Elm program to appear to lock up. Elm’s compiler can detect infinite recursion in some cases, such as “obvious” circular function calls (e.g. function `a` calls function `b`, `b` calls `c`, and `c` calls `a`, all without any form of decisions like `case` or `if` expressions), but detecting more subtle infinite recursions is not attempted.

before the bug is encountered. In Elm, we have to make this partiality explicit in the type system. We can convert this partial function into a total function using a special type called *Maybe*:

```
specialNumber n =  
  if n == 42 then  
    Just "The answer to life, the universe, and everything"  
  else  
    Nothing
```

It is now encoded at the type level that this function might not return a value for some inputs, i.e. it may return *Nothing*. We have thus converted our partial function into a total one which will always return a value. In a practical sense, in a production Elm application<sup>5</sup> there are never any circumstances that would cause an exception or a crash.

This is even more important in other types of case coverage, such as sum data types (the simple ones of which may be known as enumerated types in other languages). Consider the following data type representing the states of a traffic light:

```
type TrafficLight =  
  Red | Yellow | Green
```

From this we could try to write the following transition function to change the states:

```
changeState light =  
  case light of
```

---

<sup>5</sup>Elm does have a function `Debug.todo~::~String -> a` which will crash a program when reaching that value, but these are not allowed in optimized production applications or in published Elm packages.

```
Red -> Green
Green -> Yellow
```

Because of Elm’s totality guarantees, doing so would result in a compiler *error* whereas in Haskell it would only result in a warning<sup>6</sup>:

```
-- MISSING PATTERNS ----- TrafficLight.elm
```

This ‘case’ does not have branches for all possibilities:

```
11|> case light of
12|>   Red -> Green
13|>   Green -> Yellow
```

Missing possibilities include:

```
Yellow
```

This helps the programmer catch mistakes and even provides the helpful hint that the `Yellow` possibility is missing. Errors like these can also be used for refactoring and adding new functionality. Programmers start by encoding their change in the types, and if they have designed sufficiently instructive and powerful types, the compiler errors will provide a “treasure map” of required changes to make. This is similar to the type-directed programming approach cited in literature [63]. This is especially useful for projects which have a large number of modules or very large modules. With

---

<sup>6</sup>There is a Haskell compiler flag to enforce such warnings as errors; however, it is disabled by default.

this powerful type system, Elm programs stay robust even after undergoing very large refactoring efforts.

### 3.3.4 Elm’s Hindley-Milner Type System

Elm uses a Hindley-Milner type system, but is an extended version of Hindley-Milner compared to the one discussed in Section 3.2. This section discusses Elm’s type system’s syntax and semantics. The semantics of Elm’s type system, presented in this chapter, are useful to underpin the automatic, model-driven encoder and decoder generation presented in Section 5.2.

Definition 3.3.4 provides a mathematical definition of Elm types, adapted from [49]. Here are some important notational conventions:

- $\mathbb{N}$  denotes the set of natural numbers, starting from 0.<sup>7</sup>
- $\mathbb{N}_1 = \{n : \mathbb{N} \mid 1 \leq n\}$  denotes the set of natural numbers excluding 0.
- $\mathbb{N}_i^j = \{n : \mathbb{N} \mid i \leq n \leq j\}$  denotes the set of natural numbers from  $i$  to  $j$  inclusively.
- $\mathcal{V}$  denotes the set of all symbols.
- $\forall \alpha . T'$  represents a polymorphic type with  $\alpha$  as a type variable. In Elm, type variables are identifiers starting with lowercase letters. As in many other languages [44], we do not need to write the  $\forall \alpha$  in Elm; we simply include the type variable and the  $\forall$  is implicit. For instance, we can write `f : List a -> Maybe a` without including the  $\forall$ . In Haskell, we have the option, but not the

---

<sup>7</sup>Payr [49] uses  $\mathbb{N}$  to mean the natural numbers starting at 1, but we will include 0 for this thesis and use  $\mathbb{N}_1$  to exclude 0.

requirement, to include it. It is often included in the mathematical notation to make the type rules more clear [44].

- $\mu C$  is called a *recursive quantifier*. Using the symbol  $C$ , this is a way to write a recursive structure in a non-recursive fashion [49]. Since it's recursive, we need to ensure that every algebraic type has at least one base case; that is, that at least one of the constructors does not reference the type  $C$  itself. Thus, we require the constraint  $\exists i \in \mathbb{N}. \forall j \in \mathbb{N}_1^{k_i}. T_{i,j} \neq C$  [49]. Example 3.3.1 shows an example to illustrate this notation and property.

The following definition of Elm's type system is adapted from Payr [49]<sup>8</sup>.

#### Definition 3.3.1: Elm Types

Let Elm types be represented by  $\mathcal{T} = \{T \mid \text{is-elm-type } T\}$ , according to:

$$\text{is-elm-type } T = \text{is-mono-type } T \vee \text{is-poly-type } T \quad (3.3.1)$$

$$\begin{aligned} \text{is-mono-type } T = & \text{is-type-var } T \vee \text{is-type-app } T \\ & \vee \text{is-alg-type } T \vee \text{is-prod-type } T \\ & \vee \text{is-funct-type } T \end{aligned} \quad (3.3.2)$$

$$\begin{aligned} \text{is-poly-type } T = & T \text{ has form } \forall \alpha. T' \\ & \text{such that } (\text{is-mono-type } T' \vee \text{is-poly-type } T') \\ & \wedge \alpha \in \mathcal{V} \end{aligned} \quad (3.3.3)$$

$$\text{is-type-var } T = T \in \mathcal{V} \quad (3.3.4)$$

---

<sup>8</sup>Payr uses a less formal definition, which has been formalized here.

$$\begin{aligned}
\text{is-type-app } T = T \text{ has form } C \ T_1 \dots T_n \\
\text{such that } n \in \mathbb{N}_1 \wedge C \in \mathcal{V} \wedge \forall i \in \mathbb{N}_1^n . \text{is-mono-type } T_i
\end{aligned}
\tag{3.3.5}$$

$$\begin{aligned}
\text{is-alg-type } T = T \text{ has form} \\
\mu C . C_1 \ T_{1,1} \dots T_{1,k_1} \mid \dots \mid C_n \ T_{n,k_1} \dots T_{n,k_n} \\
\text{such that } \exists i \in \mathbb{N}_1 . \forall j \in \mathbb{N}_1^{k_i} . T_{i,j} \neq C \\
\text{where } n \in \mathbb{N} \wedge \forall i \in \mathbb{N}_1^n . k_i \in \mathbb{N} \wedge C \in \mathcal{V} \\
\wedge \forall i \in \mathbb{N}_1^n , j \in \mathbb{N}_1^{k_i} . \text{is-mono-type } T_{i,k_j} \vee T_{i,k_j} = C
\end{aligned}
\tag{3.3.6}$$

$$\begin{aligned}
\text{is-prod-type } T = T \text{ has form } \{l_1 : T_1, \dots, l_n : T_n\} \\
\text{where } n \in \mathbb{N} \wedge l_i \in \mathcal{V} \wedge \forall i \in \mathbb{N}_1^n . \text{is-mono-type } T_i
\end{aligned}
\tag{3.3.7}$$

$$\begin{aligned}
\text{is-func-type } T = T \text{ has form } T_1 \rightarrow T_2 \\
\text{where } \text{is-mono-type } T_1 \wedge \text{is-mono-type } T_2
\end{aligned}
\tag{3.3.8}$$

### Ordered Product Types (Tuples)

One thing missing from Definition 3.3.1 are ordered product types, known as tuples. Their definition is shown in Definition 3.3.2, as adapted from Payr [49]. Tuples can be considered as ordered product types where the elements are referred to by their position [49]. Here we define them in general, but Elm restricts tuples to maximum length 3.



### Definition 3.3.2: Labels and Ordered Product Types

Let  $n \in \mathbb{N}_1, T_i \in \mathcal{T}, \forall i \in \mathbb{N}_1^n, l_i \in \mathcal{V}$ .

We say that  $l_i$  is a label for the product type  $\{l_1 : T_1, \dots, l_n : T_n\}$ .

We define:

$$T_1 \times \dots \times T_n := \{1 : T_1, \dots, n : T_n\} \quad (3.3.9)$$

as the *ordered product type* with  $n$  components [49].

### Example 3.3.1: Example Recursive Definition

The following Elm type, a polymorphic tree:

```
type Tree
    = Leaf Int
    | Branch Tree Int Tree
```

can be represented formally as:

$\mu \text{ Tree} . \text{Leaf Int} \mid \text{Branch Tree Int Tree}$

This definition is legal because there exists the **Leaf** constructor which does not have **Tree** as one of its arguments.

## Elm Type Syntax

Furthermore, the type syntax in Elm is very important for the framework, so it is also included in Definition 3.3.4 as described by Payr [49].

### Definition 3.3.3: Syntax for Type Signatures

Elm's type signatures are defined using the following grammar from Payr [49], adapted into EBNF [32] form. Note that **<upper-var>** and **<lower-var>** define

uppercase and lowercase variables, respectively.

$$\begin{aligned} \langle \text{list-type-fields} \rangle &::= \langle \text{lower-var} \rangle ":" \langle \text{type} \rangle \\ &\quad \{ ", " \langle \text{list-type-fields} \rangle \} \end{aligned} \tag{3.3.10}$$

$$\begin{aligned} \langle \text{type} \rangle &::= \text{"Bool"} \\ &\quad | \text{"Int"} \\ &\quad | \text{"List"} \langle \text{type} \rangle \\ &\quad | \text{"("} \langle \text{type} \rangle ", " \langle \text{type} \rangle \text{"}")} \\ &\quad | \text{"("} \langle \text{type} \rangle ", " \langle \text{type} \rangle ", " \langle \text{type} \rangle \text{"}")} \\ &\quad | \text{"{"} \langle \text{list-type-fields} \rangle \text{"}" } \\ &\quad | \langle \text{type} \rangle \text{"->" } \langle \text{type} \rangle \\ &\quad | \langle \text{upper-var} \rangle \{ \langle \text{type} \rangle \} \\ &\quad | \langle \text{lower-var} \rangle \\ &\quad | \text{"("} \langle \text{type} \rangle \text{"}")} \\ &\quad | \text{"("} \end{aligned} \tag{3.3.11}$$

#### Definition 3.3.4: Syntax for Type and Type Alias Definitions

Elm's types are defined by `type` and `type alias` statements. The following definition is adapted from Payr [49]. Note that `<upper-var>` and `<lower-var>`

define uppercase and lowercase variables, respectively.

$$\langle \text{list-type-constr} \rangle ::= \langle \text{upper-var} \rangle \{ \langle \text{type} \rangle \} \quad (3.3.12)$$
$$[ " | " \langle \text{list-type-constr} \rangle ]$$
$$\langle \text{type-statement} \rangle ::= \text{"type"} \langle \text{upper-var} \rangle \{ \langle \text{lower-var} \rangle \} \text{"="}$$
$$\langle \text{list-type-constr} \rangle$$
$$| \text{"type alias"} \langle \text{upper-var} \rangle \{ \langle \text{lower-var} \rangle \}$$
$$\text{"="} \langle \text{type} \rangle$$
$$(3.3.13)$$

### 3.3.5 The Elm Architecture: Model-View-Update (MVU)

As discussed in Section 3.3.1, Elm uses a standard architecture for all programs, known as The Elm Architecture, or TEA. TEA, also known as Model-View-Update (or MVU) is an event-driven architecture, consisting of four main components: the model, messages, view function, and update function<sup>9</sup>.

At a high level, The Elm Architecture is an event-driven paradigm where the `update` and `view` functions are callback functions for updating the state and the rendered output, respectively. They can be understood via the dataflow diagram in Figure 3.1. Unlike traditional EDP, where the user has to manage the complexity of registering events and callbacks (of which there could be an arbitrary number), it is the Elm runtime’s responsibility to process events and call the appropriate functions

---

<sup>9</sup>Two other components, commands and subscriptions, are also available for advanced applications, which are discussed in the following section.

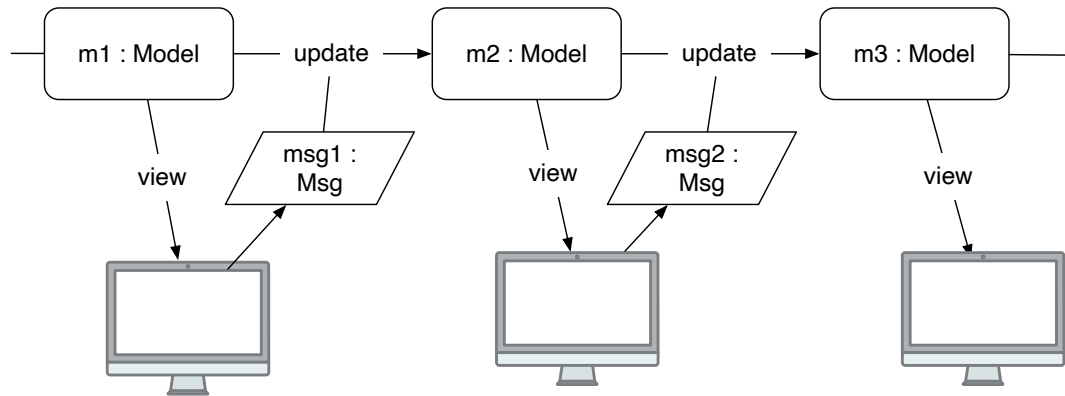


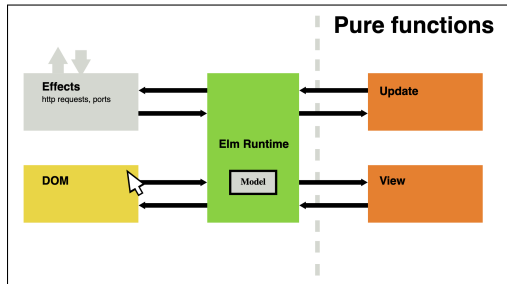
Figure 3.1: Model-View-Update (The Elm Architecture) dataflow diagram. Models are passed into the `view` function for rendering, and messages coming from the user’s interactions are processed by the `update` function, producing a new model.

(see TEA’s lifecycle in Figure 3.2); the user’s only responsibility is to provide these functions with a known interface.

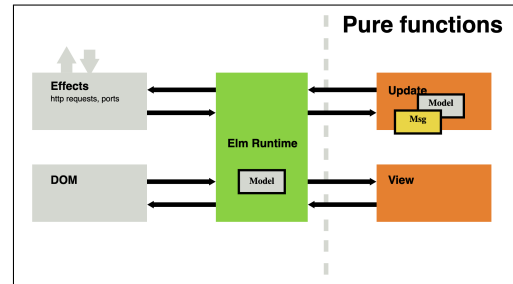
## Model

The Model is a data type defined by the user, which represents all the data that needs to change throughout the lifetime of the Elm program. The model can be defined in two ways: as a `type` declaration or as a `type alias` declaration. The former creates an algebraic datatype, also known as a sum data type, e.g. the traffic light data type in Section 3.3.3. The other way, creating it as a `type alias` declaration, allows the model to be equated to another data type, including simple data types like strings or integers, or, most commonly, a product type in the form of a record. It is common to nest a sum type inside the product type.

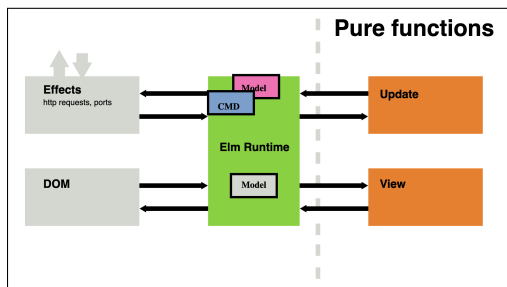
The model as a product type has the advantage of allowing for easy expansion as the program is modified. Unfortunately, product types have the drawback that it becomes very easy for programmers to create data types with much larger cardinalities



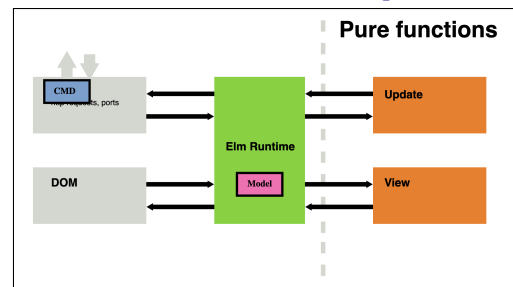
(a) The user clicks a button, causing a message to be sent.



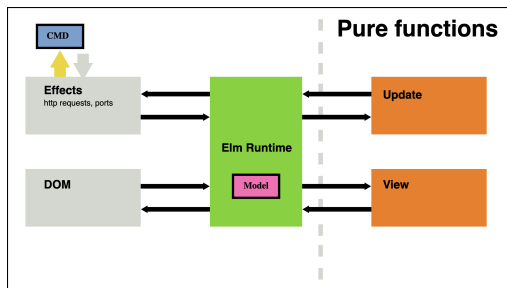
(b) The Elm runtime passes the message and the current model into the `update` function.



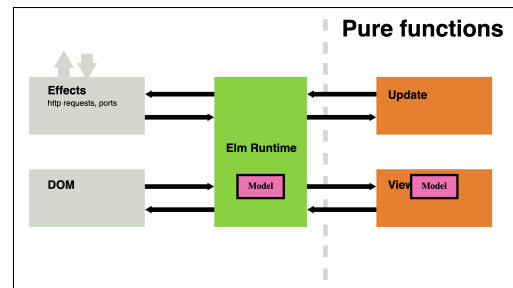
(c) The update function returns a new model and commands to be processed.



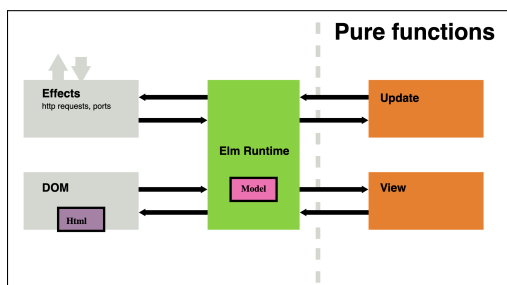
(d) The runtime stores the model and sends the command to the effects processor.



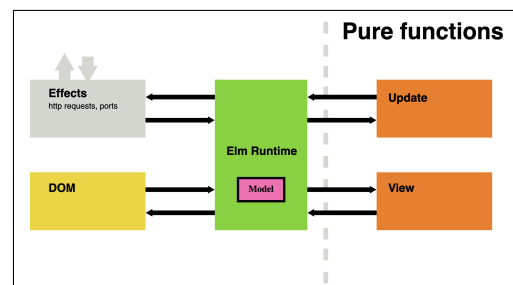
(e) The effects processor begins processing the command.



(f) The new model is passed into the `view` function.



(g) The view function passes back HTML to be rendered.



(h) The Elm runtime updates the Document Object Model (DOM).

Figure 3.2: The lifecycle of the Elm Architecture.

than is reasonable for the program they are modelling. This is especially true of novice programmers and those who have little background in discrete mathematics or database normalization.

## Messages

Messages, which are represented by a type or type alias called `Msg`, are discrete actions (events) which can be processed by the Elm runtime. Unlike for the model, messages are most naturally represented as sum (algebraic) data types, with one constructor for each action that could happen in the program. Messages can be sent in many ways, including from the `view` function, as a result of a processed command, or as a notification from a subscription (see Section 3.3.5). Messages are processed by the `update` function to produce a new model.

## View

The `view` function is a function which takes as input the current value of the model, and produces as output a concrete representation of the program called the view. This output can be in one of many forms, the most common being an HTML representation, or in the case of the `GraphicSVG` library, an SVG representation. The Elm runtime is responsible for performing the necessary manipulations to change what is actually rendered in the browser window.

## Update

The `update` function is a pure function which processes messages to produce a new model. Thus, it takes as input the message and the current model and produces

a new model. It processes a single message at any given time. That is, messages are atomic and are not processed concurrently. The Elm runtime is responsible for storing the current model, calling the `update` function when a message is sent, and then subsequently calling the `view` function with the new `Model` value.

#### Definition 3.3.5: The Elm Architecture (Simplified)

Elm programs are represented by the following values, provided here with their corresponding types:

- `type alias Model = ...` — A data type representing the data model; that is, the values that can change as the program executes.
- `type Msg = ...` — A data type representing the messages (actions) that can occur while the program runs. These can come from user actions or task completion (see 3.3.6).
- `init : flags -> Model` — The initial model of the program (flags can be passed in from the program runner, i.e. JavaScript).
- `view : Model -> Html Msg` — A pure function turning the current model into an HTML representation that can send messages of type `Msg`.
- `update : Msg -> Model -> Model` — A pure function that returns a new model, advancing forward the state of the application.

#### Example 3.3.2: Simple Counting Program

The following example is a simple program that allows the user to increment and decrement a counter:

```
type alias Model =
    { count : Int }

type Msg
    = Increment
    | Decrement

update : Msg -> Model -> Model
update msg model =
    case msg of
        Increment ->
            { model | count = model.count + 1 }
        Decrement ->
            { model | count = model.count - 1 }

-- myShapes is an alias for view
myShapes : Model -> List (Shape Msg)
myShapes model =
    [
        text (String.fromInt model.count)
        |> centered
        |> filled black
        |> move (0, -3)
        , button green "+"
        |> move (0, 30)
```



```
        |> notifyTap Increment
    , button red "-"
        |> move (0, -30)
        |> notifyTap Decrement
    ]
```

### Notes

- The `Model` contains one field, a integer count.
- The `Msg` contains an increment and decrement message. The `Tick` message (used for animation and keyboard input) is not used and is thus omitted.
- The `update` function returns a new model, which takes the current count and increments or decrements it.
- The `view` function renders buttons and the count, shown in Figure 3.3. It uses the `notifyTap` function to send a message when the user clicks on the increment or decrement buttons.

## Interacting with Impure Values

Since Elm is a pure language, it has special mechanisms to interact with impure values. These include things like performing an HTTP request, generating random values, and sending messages over a WebSocket connection. To achieve this, Elm provides a mechanism called *Commands*, which are added onto the output type of the `update` function in advanced applications. Commands are initiated by the user, processed by Elm's runtime asynchronously, and then results are sent back as messages to the

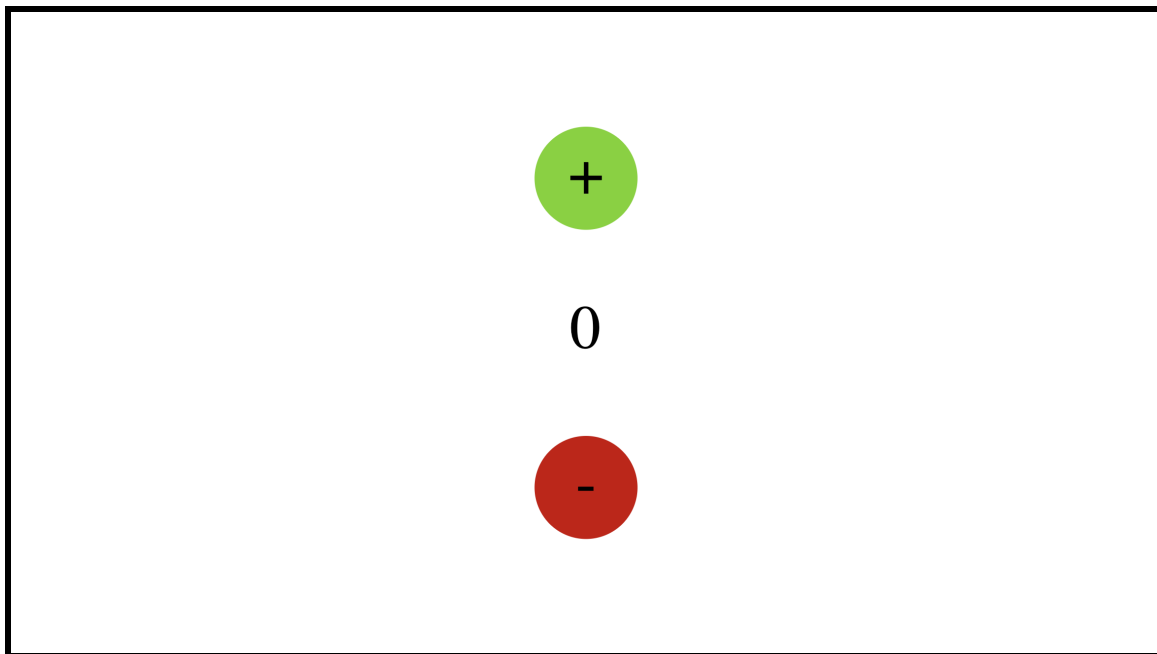


Figure 3.3: Counting example interface. The code for this example is given in Example 3.3.2.

user’s `update` function. Commands can be batched to trigger more than one at a time. Figure 3.2 shows the full lifecycle of the Elm Architecture, including how commands are processed.

For actions which could be initiated without the user interacting with the program, including timer events, browser window resize events, and *received* WebSocket messages, Elm supports the notion of *subscriptions*. Subscriptions allow the `update` function to receive messages when these events happen. Like commands, subscriptions can be batched.

Definition 3.3.6 lists the new type signatures for these advanced applications. Note the inclusion of `Cmd Msg` outputs (which means “Commands which send messages of type `Msg`”), and the new `subscriptions` function, which returns a `Sub Msg`, or a subscription producing messages of type `Msg`. The `Model` input to the `subscriptions`

function allows the programmer to turn subscriptions on or off based on the state of the model.

### Definition 3.3.6: The Elm Architecture (Advanced)

Elm programs are represented by the following values, provided here with their corresponding types:

- `type alias Model = ...` — A data type representing the data model; that is, the values that can change as the program executes.
- `type Msg = ...` — A data type representing the messages (actions) that can occur while the program runs. These can come from user actions or task completion (the result of commands or subscriptions).
- `init ~:~ flags -> (Model, Cmd Msg)` — The initial model of the program (flags can be passed in from the program runner, i.e. JavaScript). This function also supplies a command to be processed when the program launches (which can be `Cmd.none`, meaning “do nothing”).
- `view : Model -> Html Msg` — A pure function turning the current model into an HTML representation that can send messages of type `Msg`.
- `update : Msg -> Model -> ( Model, Cmd Msg )` — A pure function that returns a new model, advancing forward the state of the application, and a command to be processed (which can be `Cmd.none`, meaning “do nothing”).
- `subscriptions : Model -> Sub Msg` — A function which returns zero or more *subscriptions*, events to listen for which produce messages.

## 3.4 Haskell

Haskell is a purely functional, statically typed language with lazy evaluation [25]. It uses a Hindley-Milner type inference system, similar to Elm, but with many additional features, like user-defined type classes, higher-rank polymorphism, and support for basic dependent types.

The Haskell language was used to create the server backend for TEASync, as well as the online Integrated Development Environment (IDE) in which programmers can collaboratively write, test, and deploy their applications. This section provides a brief overview of Haskell as well as some key frameworks used in the development of TEASync.

### 3.4.1 History

The Haskell language has its origins in 1987 at the FPCA conference in Portland, Oregon, where a committee was formed to combine the features of several disparate languages into one. In 1990, the first Haskell Report was published, motivation, nature and process of the creation of Haskell. In 1992, the first tutorials for writing Haskell, as well as the GHC compiler (the most widely-used Haskell compiler) were created [46].

### 3.4.2 Integrated Haskell Platform (IHP)

The Integrated Haskell Platform (IHP) was created by digitally induced GmbH and released publicly in 2020. It is a “batteries-included” web framework, combining data modelling, server-side rendering, and database communication [16]. It follows

a model-view-controller architecture, which, like TEA, separates data from the rendering and rules for updating the data. Unlike TEA, the controller is not pure and handles side effects as IO operations. IHP was chosen for its ease of use, built-in WebSocket support, type safety, and overall reliability.

### 3.4.3 Software Transactional Memory (STM)

Haskell is a concurrent language [25], and provides support for creating extremely lightweight threads to perform tasks in parallel. Software transactional memory (STM) is a flexible, non-blocking approach to synchronizing data across threads. The word transactional implies operations that are atomic, running through to completion before another one can start. Empirical evidence shows that this approach outperforms other lock-free translation methods [54]. The Haskell language provides a library called `Control.Concurrent.STM` [45] which we use in the current work.

Of particular interest to the current work is STM’s handling of transactional queues (called `TQueue`). `TQueues` are thread-safe first-in, first-out message queues, which simplify the communication between threads, requiring no user-defined locks, semaphores, etc. The important functions related to `TQueues` are shown in Definition 3.4.1.

#### Definition 3.4.1: STM `TQueues`

Below are some basic functions for the `TQueue` data type [45]:

- `data TQueue a` — A data type representing a transactional queue with values of type `a`.
- `data STM a` — An data type representing an atomic action which returns

a value of type `a`.

- `atomically :: STM a -> IO a` — Perform an STM action (e.g. one of the following functions) atomically, as part of an IO operation.
- `newTQueue :: STM (TQueue a)` — Create a blank transactional queue which can handle values of type `a`.
- `writeTQueue :: TQueue a -> a -> STM ()` — Write a value of type `a` into a transactional queue. Writes happen instantly and do not block.
- `readTQueue :: TQueue a -> STM a` — Read the next value of type `a` from a transactional queue. Blocks execution until a value arrives.

## Chapter 4

# TEASync Framework Architecture

This chapter provides an overview of the design process of the TEASync framework’s architecture. The discussion begins with the “fundamental theorem of TEASync”, which describes how the idea for the framework came about, and how Elm’s MVU pattern and its properties lends itself well to a multi-user framework. This is followed by a detailed overview of the Local-Global Model-View-Update (LG-MVU) model and its components. Then we discuss the concurrency design of the system, including a discussion of optimistic vs. pessimistic model updates and how to balance correctness and responsiveness. Next, we provide an overview of alternative synchronization schemes that were considered, and how each one maintains *eventual consistency* amongst the clients. We also discuss the reasons for the default choice of message-based synchronization with folding. Finally, we present mathematically desirable properties for designing apps that further minimize ill effects of race conditions.

## 4.1 “Fundamental Theorem of TEASync”

Theorem 4.1.1 is the idea on which the TEASync framework and the LG-MVU architecture is built. We will return to this theorem throughout this chapter and the remaining chapters as we explain 1) why this theorem is important to create a multi-user application, 2) why Elm and its Elm Architecture is a natural fit to achieve this, and 3) how TEASync provides the remaining necessary technology to allow the ordering of messages sent by clients to remain intact and be broadcast to all other clients.

### Theorem 4.1.1: Fundamental Theorem of TEASync

Given  $n$  TEASync clients with identical `update` functions, `Model` types and values for `init`, and an identical ordering of messages, the final value for the `model` will be identical.

Fundamentally, this theorem states that if identical clients process the same messages in the same order, then they will remain *synchronized*, and thus the application will be a multi-user application.

Elm and The Elm Architecture’s properties of pureness and atomicity make this easy to achieve from the client-side perspective. An Elm application can be viewed as a long-running *fold* over a sequence of messages, with `init` as the initial value, `update` as the higher-order function, and the final `model` as the output. Elm’s pureness for functions is a key part of why this theorem holds given Elm as the client language. While the TEASync method would be realizable in any language, it would only be able to do so if the `update` function were constrained by the language or manually checked to be pure. Manually checking would need to verify that the function’s result is only a consequence of the inputs to the function, that such inputs were pass by



value, and that the function does not modify any memory beyond its local scope.

Furthermore, Elm’s data-based event representation (messages) makes encoding and decoding values for transport through a server a natural extension. All that remains is the need for a server to ensure that all clients receive messages in the same order, which we will explore later in this chapter.

## 4.2 Local-Global Model-View-Update (LG-MVU) Architecture

We present a new extension to MVU, called Local-Global Model-View-Update (LG-MVU), in which the model is split into a client-specific *local* portion, and a shared *global* portion. This section will justify the need for LG-MVU then discuss the data flow and components of LG-MVU.

### 4.2.1 Motivation

It would be possible to create a framework using the usual MVU, but with a server backend to synchronize the model. We will refer to this as the *shared MVU* architecture. This would indeed keep all clients in-sync and satisfy the constraints laid out by the Theorem 4.1.1. However, it is not always advantageous to synchronize *all* the state amongst all clients. Many applications beyond the most trivial ones have some state that is local to the client in question. In *shared MVU*, the programmer would need to use complicated workarounds like keeping client-specific state in a dictionary, and even then it would be necessary to have a client ID specific to each client.

LG-MVU allows the programmer to have a clear separation of state needed for each

client and the state that needs to be shared. We hypothesize that this will not only be easy for programmers to reason about, but that it will force newer programmers to learn how to design applications with a clear separation of state. Determining if this is true was one of the goals of the focus groups and surveys.

### 4.2.2 Components

LG-MVU contains several components, which are similar to MVU's components but with a split between local and global models and messages. Definition 4.2.1 lists all the components of the simplified form of LG-MVU.

#### Definition 4.2.1: Local-Global Model-View-Update (Simplified)

LG-MVU programs are represented by the following values, provided here with their corresponding types:

- `type alias LocalModel = ...` — A data type representing the local data model; that is, values that can change locally on the client.
- `type LocalMsg = ...` — A data type representing the local messages (actions) that can occur while the program runs. These can come from user actions or task completion (see 3.3.6), and are not shared with the other clients.
- `initLocal : LocalModel` — The initial local model of the program. Flags are omitted from this simplified version.
- `localUpdate : LocalMsg -> LocalModel -> GlobalModel -> LocalModel` — A pure function that returns a new local model, in

response to a local message. The global model is also passed in as an argument for convenience. Thus, the new local model is a function of the local message and the current local and global models.

- `type alias GlobalModel = ...` — A data type representing the global data model; that is, values that can change and are synced with all clients.
- `type GlobalMsg = ...` — A data type representing the global messages (actions) that can occur while the program runs. These can come from user actions or task completion (see 3.3.6), and are shared with the other clients, keeping all clients synchronized.
- `initGlobal : GlobalModel` — The initial global model of the program. Flags are omitted from this simplified version.
- `globalUpdate : GlobalMsg -> GlobalModel -> GlobalModel` — A pure function that returns a new global model, in response to a global message. Note that a local model is not an argument, since this could easily lead to inconsistencies amongst clients. Thus, the new global model is a function of the global message and the current global models.
- `view : LocalModel -> GlobalModel -> Html (TEASync.Msg LocalMsg GlobalMsg GlobalModel)` — A pure function turning the current model into an Html representation that can send messages of type `TEASync.Msg` (which has constructors for `LocalMsg` and `Global`). Thus, this `view` function can send both local and global messages.

## Local Model and Messages

The *local* portion of LG-MVU is a model and messages that control actions specific to the given client, i.e. not to be shared with other clients. This is often helpful for UI actions specific to the client. One canonical example of this is buttons reacting to the user’s mouseover. Not only is this (usually) unnecessary to share, but this is an action that happens quite frequently and would cause extra load on the server if shared unnecessarily. It is also an action where the user expects instant feedback, so keeping this local to the client makes sense (see Section 4.3.1 for a discussion on increasing the perceived responsiveness of global messages).

## Global Model and Messages

The *global* portion of LG-MVU is a model and messages that control shared actions and state. As evidenced by the name, the global model and messages are used for actions and data that should indeed be shared amongst clients.

## View

The `view` function takes in both the local and global models as input and has the ability to produce both types of messages.

### 4.2.3 Data Flow

Similar to MVU, we can represent LG-MVU’s dataflow over the lifetime of a running application using a dataflow diagram, as shown in Figure 4.1. The diagram shows an example LG-MVU application with two clients. The global model represents the shape displayed on the (tiny) screen, and the local state represents the colour. The

user can change the colour of their shape independently of the other users, but if a user changes the shape, this is propagated to all clients. The `view` function is a function of both the local and global models, and can send local and global messages.

## 4.3 Concurrent System Design

Care must be taken to ensure that TEASync, as a concurrent system, both scales effectively and maintains correctness to satisfy the ordering property of Theorem 4.1.1. This section will explain how this was achieved.

Figure 4.2 shows a sequence diagram of the lifecycle of a TEASync application (using the default message-based synchronization scheme with folding). Since Elm only processes one message at a time, the client can be viewed as its own thread processing each message in the queue atomically. Each client has a corresponding send and receive thread on the server, responsible for encoding/decoding and sending/receiving messages. The main thread is responsible for spawning the child threads, but for all subsequent operations, the send and receive threads communicate directly. This allows the system to scale and take advantage of multi-core hardware.

To ensure ordering guarantees, the server is the ground truth for message ordering. Communication amongst threads is done using atomic queues, so all clients receive messages in the same order. The server-imposed message order is shown as the global message being annotated with a  $k$  variable only after being processed by the receiving thread. The transport method between server and client must also maintain FIFO ordering.

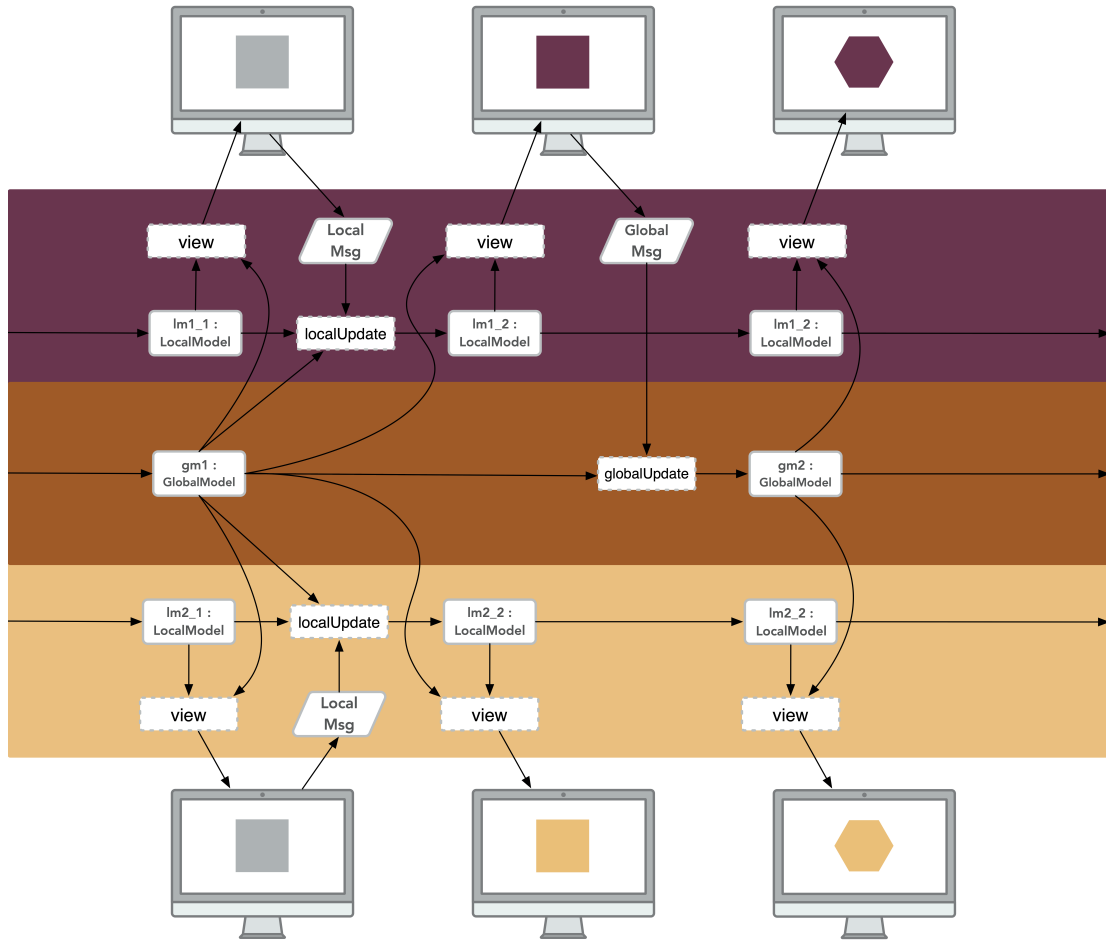


Figure 4.1: A two-client dataflow diagram of the Local-Global Model-View-Update (LG-MVU) Architecture. Time proceeds from left to right, with arrows representing the flow of data between components. Rectangles with solid outlines are instances of the data. Each client has its own local model (top and bottom band), and all clients share a global model (middle band). In this application, the global state represents the shape displayed on the screen and the local state represents the colour.

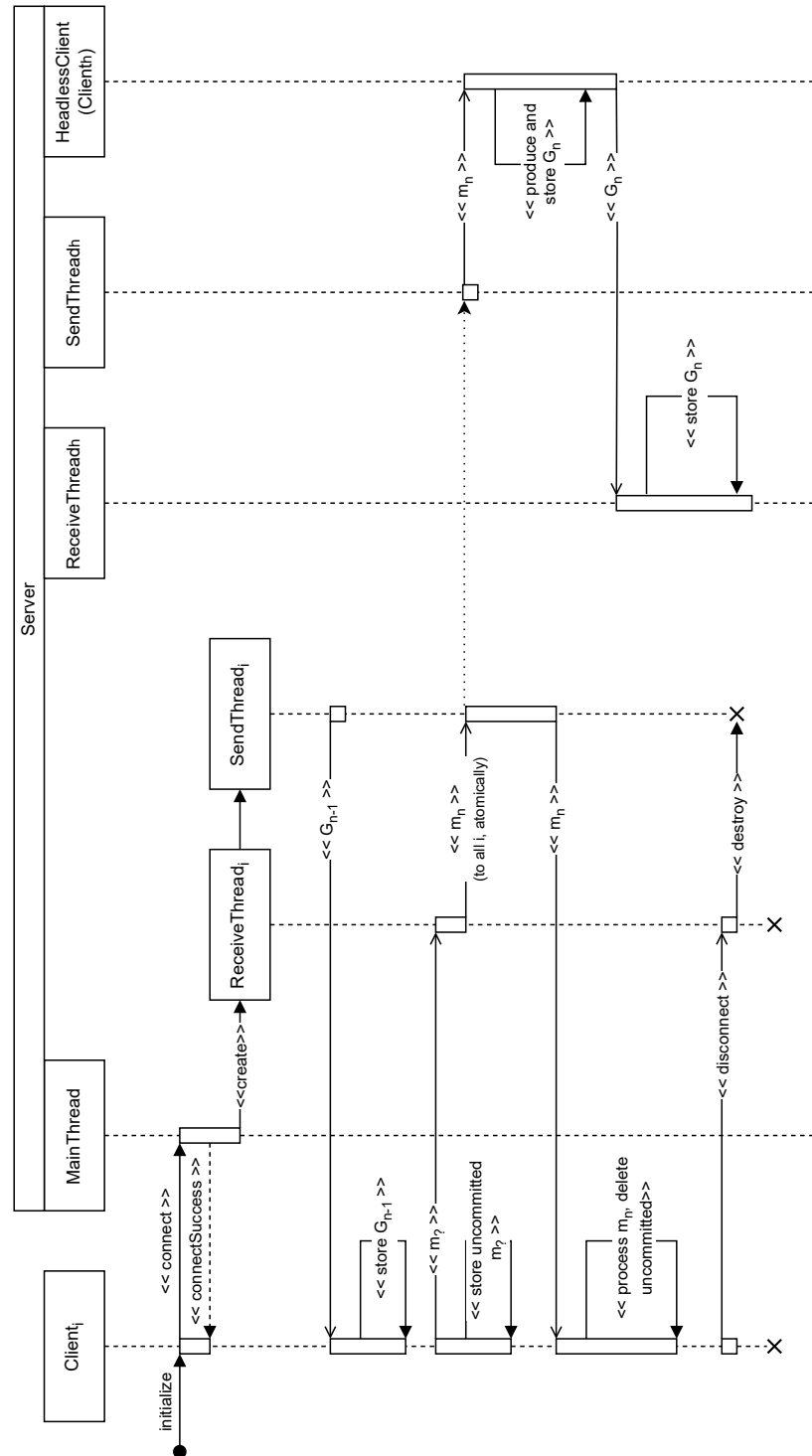


Figure 4.2: Sequence diagram showing the lifecycle of a TEASync client connecting, sending a global message, and disconnecting, using the default message-based synchronization scheme with folding.

### 4.3.1 Optimistic vs. Pessimistic Model Updates

Local messages can be processed right away since their ordering (and, indeed, content) has no effect on the state of other clients. However, for global messages there is a tension between responsiveness and accuracy, often referred to as optimistic versus pessimistic rendering. In optimistic rendering, the message is processed right away on the client before receiving the message back from the server, whereas pessimistic rendering would wait to get the message back from the server before updating the screen. The latter obviously incurs a responsiveness penalty as client requires a round trip to and from the server prior to updating the screen. However, the former would be in violation of the ordering property of Theorem 4.1.1.

TEASync uses an hybrid strategy that combines optimistic and pessimistic rendering. Upon sending a new global message, the client in question tags the message with its client ID, which is sent back from the server alongside each message. The sending client stores the message in a queue of uncommitted messages. This list of uncommitted messages is applied to the stored version of the model prior to passing it into the `view` function. This has the effect of immediately showing the user the result of the action, even before the round trip is complete. Once the message is received by the client, it is identified via the identifier, removed from the uncommitted list and is used to update the client’s copy of the global model as usual.

This strategy means that the sending client’s view is said to be *eventually consistent* with the other clients. This can lead to temporary inconsistencies in the view, particularly if the global update function and messages do not adhere to specific properties, such as idempotency and associativity (see Section 4.5).



## 4.4 Alternative Synchronization Schemes

Five different synchronization schemes were considered: global model-based (MoB), global model-diff-based (MoDB), message-based (MsB), message-based with folding (MsBF), and message-based with distributed folding (MsBDF). This subsection will explain, compare and contrast each of these schemes. In the end, message-based synchronization with folding was chosen as the default scheme, due to its simplicity and good concurrent properties, but message-based with distributed folding would be appropriate for less mission-critical systems (such as in-development servers) and when server resources such as memory and processing are limited compared to network bandwidth.

The implementation and run-time complexity of these schemes depends on whether immutable data structures are used, of the type supported by Elm (and often originally implemented in Haskell). With such data structures, modified dictionaries, lists and records are stored as new tree types with references to all or large parts of previous incarnations. As a result, it is cost-efficient to make small changes, keep a list of past states and roll back when necessary. The cost for almost all operations will be immediately low (because of the lack of copying), paid instead in the form of garbage collection.

Throughout many of these schemes, the example message sequence in Table 4.1 will be used to analyze how the different schemes handle concurrency problems, like race conditions.

For each case, we will give examples highlighting the strengths and weaknesses of that method.

For the following sections, let  $\star$  be a function (used as an infix function) defined

Table 4.1: Increment/Decrement Example Sequence

Time (s)	Client 1	Client 2	Desired Model
0	Increment		1
0.5		Decrement	0
1	Decrement	Decrement	-2
1.5		Increment	-1
2		Decrement	-2

Table 4.2: Comparison of Synchronization Methods

Method	Scalability	Race Condition Chance	Implementation Complexity	Communication Overhead
MoB	Moderate	High	Low	High
MoDB	Low	Moderate	High	Low
MsB	Moderate	Low	Low	Low
MsBF	High	Low	Moderate	Low
MsBDF	Highest	Low	Moderate	Moderate

Legend:

MoB = Global Model-Based

MoDB = Global Model-Diff-Based

MsB = Message-Based

MsBF = Message-Based with Folding

MsBDF = Message-Based with Distributed Folding

as the composition of two calls to a given `globalUpdate` function, one with each inputted message:

$$\star : \text{GlobalMsg} \rightarrow \text{GlobalMsg} \rightarrow \text{GlobalModel} \rightarrow \text{GlobalModel}$$

$$m \star n = \text{globalUpdate } m \circ \text{globalUpdate } n$$

### **Global Model-Based Synchronization (MoB)**

The global model-based synchronization scheme involves sending the updated global model to every other client upon each message being sent. This is perhaps the most naive method of synchronizing the global model, but it has some obvious downsides. The first is that the model usually contains a lot of data compared to the messages, thus sending the model each time would be expensive. Figure 4.3 shows the sequence of actions that happen with this type of synchronization.

The other obvious downside is that this makes the chance of race conditions very high; that is, the chance of one client's updates overwriting another's is very high, due to the random ordering of messages compounded by differences in round trip message times between each client and the server. This is the factor that makes this scheme ultimately extremely undesirable.

Figure 4.4 shows how this scheme handles the sequence of messages in Table 4.1, illustrating the issue with this type of synchronization. At time  $t = 1s$ , there is a race condition since both clients decrement at the same time, sending each other the message that the model should be `{ count = -1 }`. This effectively erases one of the clients' decrement actions.

### **Global Model-Diff-Based Synchronization (MoDB)**

Like global model-based synchronization, this scheme shares data by updating the model directly, but with diffing to only send updates of the parts of the model that were updated by the update function. In this scheme, the client would compute the difference between the new model and the old model (or alternatively, keep a ledger of the changed parts), and determine which parts of the model to send updates for.

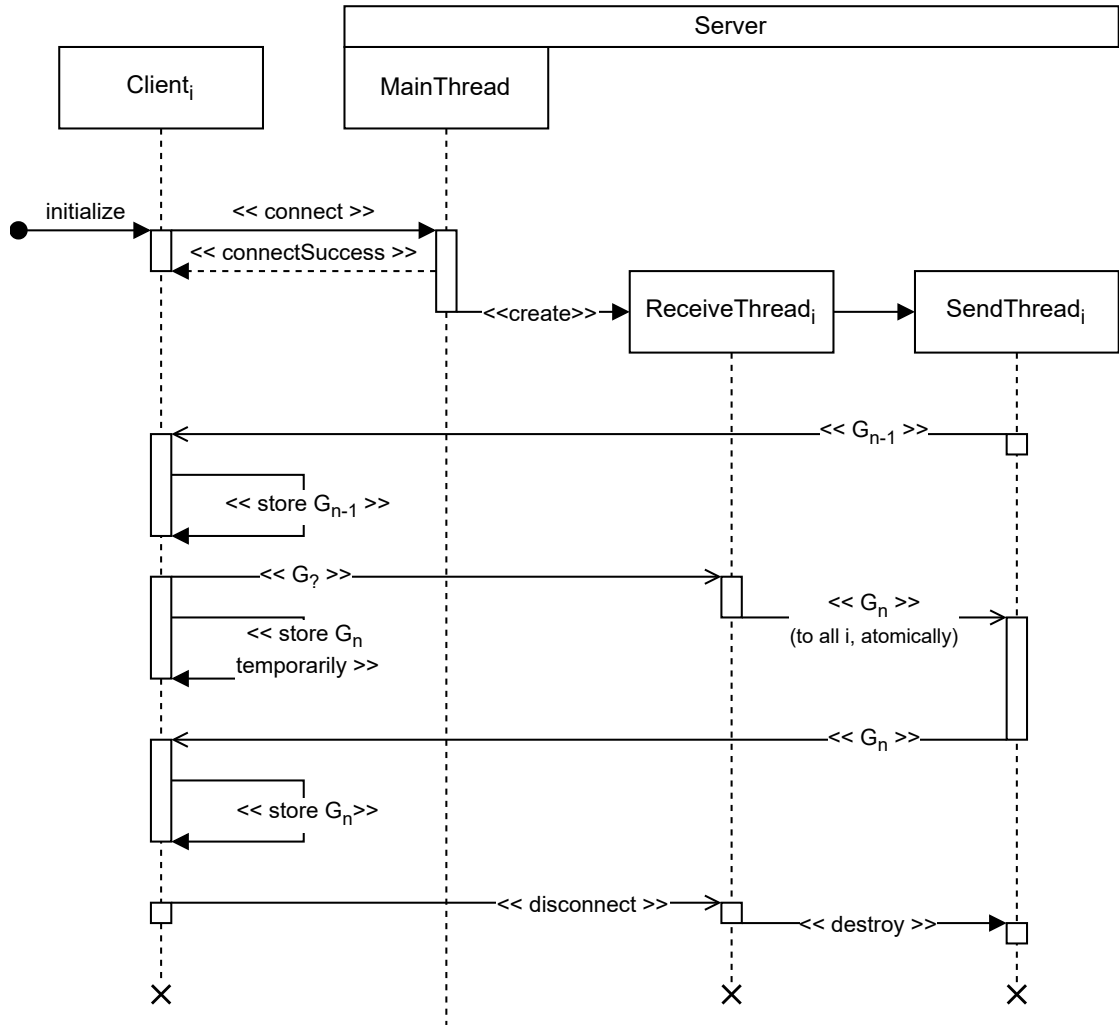


Figure 4.3: Sequence diagram showing the lifecycle of the global model-based synchronization method. This method sends the global model each time a client updates the global model, which has the downside of both network traffic and an increased likelihood of race conditions.

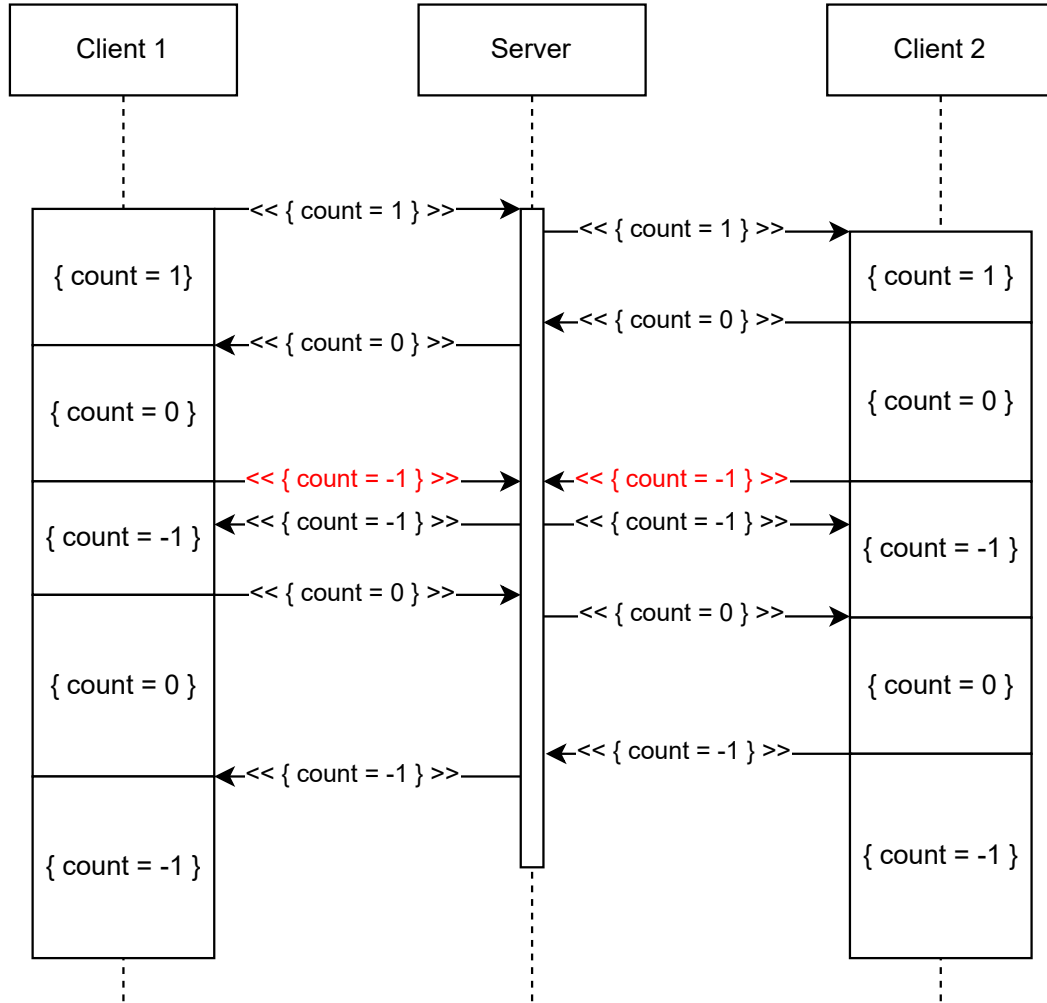


Figure 4.4: Sequence diagram showing how the global model-based synchronization scheme handles the sequence of messages in Table 4.1. At  $t = 1s$  there is a race condition that causes the two clients' decrements to cancel out (shown in red).

Our simple counting example contains only one field, and thus in this simple example, it is very similar from the Model-Based synchronization approach. For more complex models, this would reduce communication overhead and the chance of race conditions but at the expense of complexity. And the chance of race conditions is still higher than other schemes, like the model-based scheme.

Figure 4.5 shows how this scheme handles the sequence of messages in Table 4.1. Even though the clients only send updates about the parts of the model that were modified, there is still a race condition. This illustrates how this scheme only improves upon the regular model-based one if messages operate on different parts of the global model.

### **Message-Based Synchronization (MsB)**

In pure message-based synchronization, global messages are sent by the client and are stored in order on the server. Each time a client connects, the server sends all the stored messages, and the client plays “catch-up” by applying all the messages to the initial global model. This scheme is able to reduce race conditions and has a relatively low implementation complexity but these come at the expense of network traffic and client-side processing time. A long-running application may process thousands of messages, or more! This would obviously cause a large startup cost for connecting clients, that only gets worse the longer the application is running.

### **Message-Based Synchronization with Folding (MsBF)**

In this scheme, to avoid the need for the client to process all  $m$  messages that have been sent prior, the global model is sent to the client at the beginning, effectively

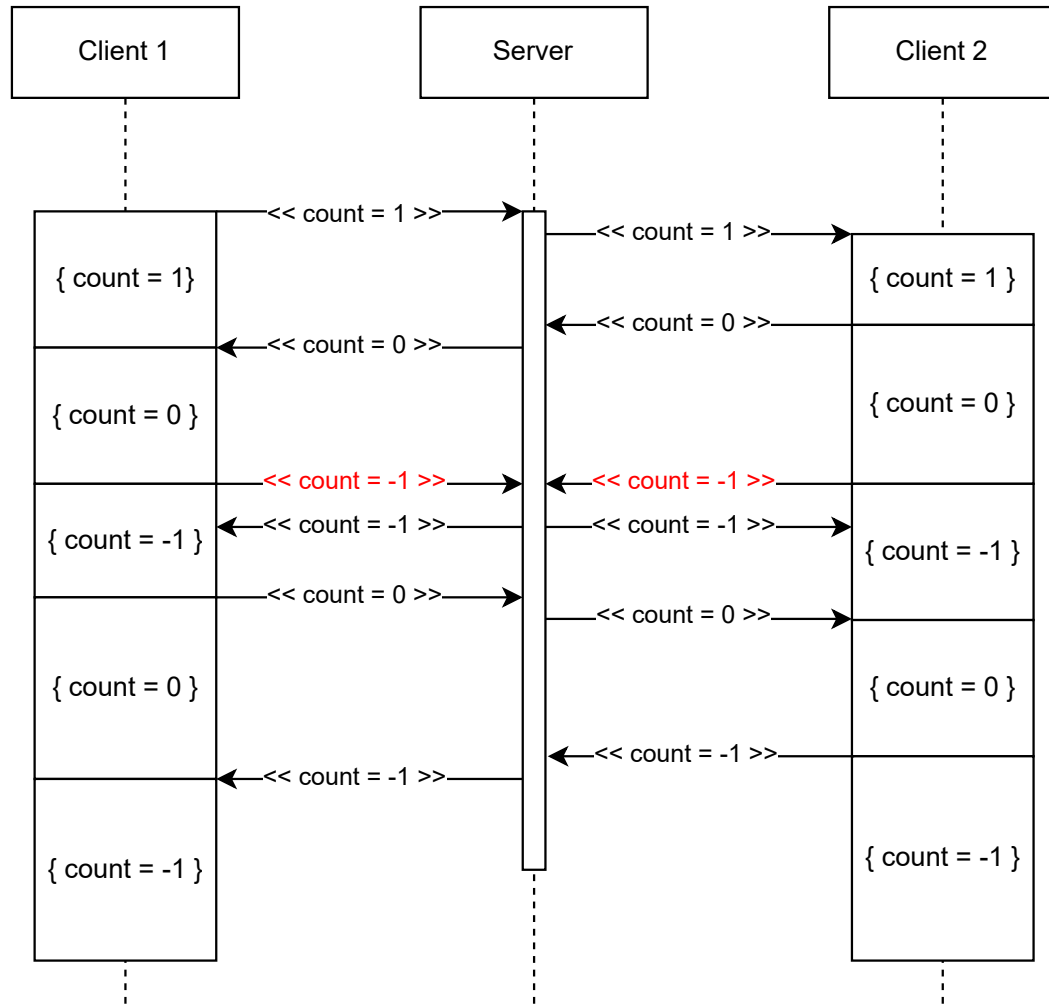


Figure 4.5: Sequence diagram showing how the global model-diff-based synchronization scheme handles the sequence of messages in Table 4.1. At  $t = 1s$  there is a race condition that causes the two clients' decrements to cancel out (shown in red). This is similar to the situation in Figure 4.4, since the two messages operate on the same field of the model.

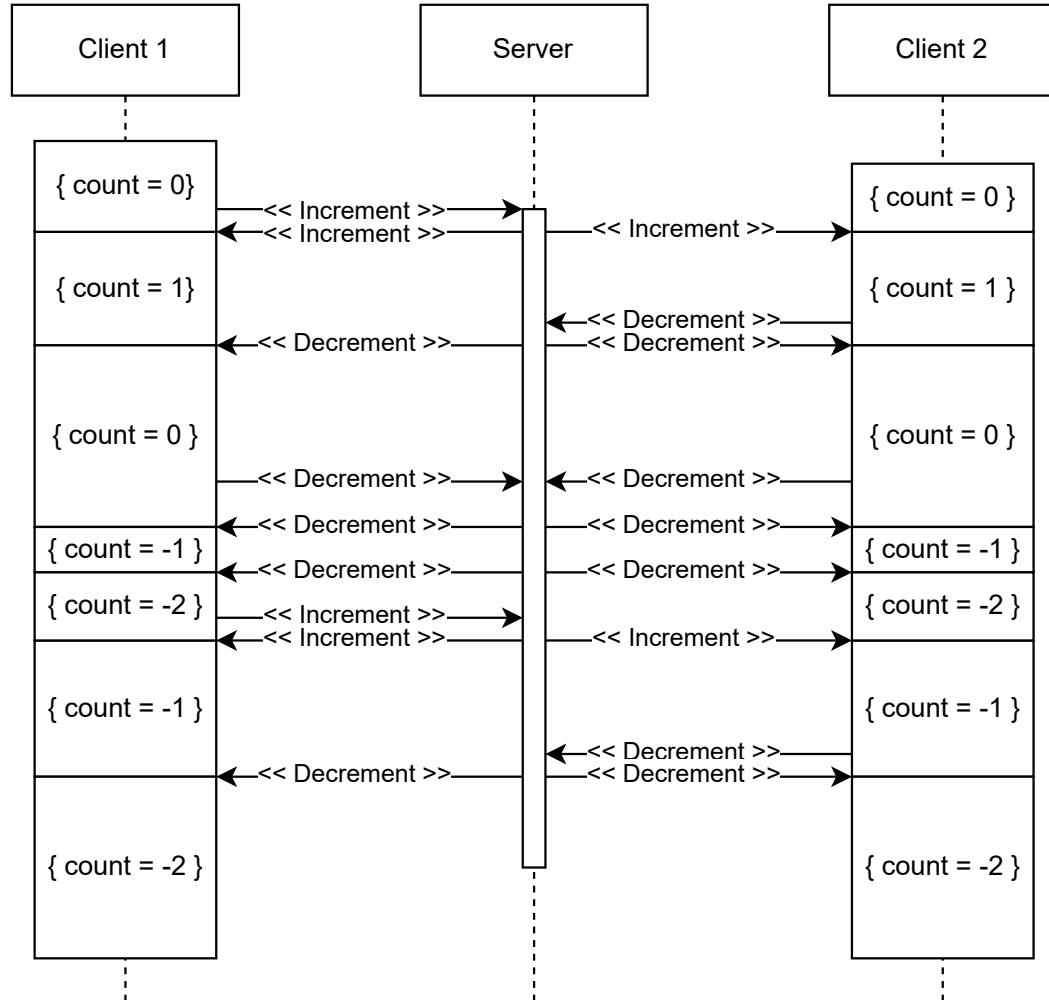


Figure 4.6: Sequence diagram showing how the message-based synchronization scheme handles the sequence of messages in Table 4.1. This scheme avoids the race condition at  $t = 1s$  since the messages are both applied and neither is lost. This is further aided by the fact that the global update function in this case has the *symmetry* property with respect to the **Increment** and **Decrement** messages (see Section 4.5) and they can thus be applied in any order.



folding (or “fast-forwarding”) the app’s folded state to the present value. This is possible since, the global model is, as in MVU, a folded representation of the messages sent before.

To specify this more formally, let  $G_0$  represent the initial model and  $G_i$  represent the global model after processing  $i$  messages (which we’ll call  $m_1, \dots, m_i$ ), thus  $G_i = (m_1 \star m_2 \star \dots \star m_i) G_0$ . At any point you could divide the folded state up into a partially-folded form: a model and a list of messages succeeding it, for instance  $G_{n-3}, m_{n-2}, m_{n-1}, m_n$ , such that  $G_n = (m_{n-2} \star m_{n-1} \star m_n) G_{n-3}$ .

In cases where the size of the model is much less than the size of the messages times the number of past messages, this is advantageous from a network perspective, and it is always advantageous from a client processing time perspective<sup>1</sup>. After that, only the global messages are sent to the clients, and all clients are all responsible for updating their own copy of the global model.

To ensure proper ordering even for the client that sent the global message, the client must wait to receive its own message from the server before processing it. Section 4.3.1 discussed how to improve responsiveness despite this challenge.

### Message-Based Synchronization with Distributed Folding (MsBDF)

To eliminate the need for a headless version of the client, this scheme has the clients send the global model back to the server when a new one is produced. Doing so would incur the same network performance penalty as model-based synchronization, which is undesirable. Figure 4.7 shows the lifecycle of a client using this scheme.

The amount of extra network traffic of a model of  $m$  bytes in a system with  $n$

---

<sup>1</sup>In general, the model will be smaller than the messages that generated it; especially if most messages represent modifications rather than insertions. Messages that delete data from the model would swing the advantage even further in the favour of the model.

connected clients would be  $\mathcal{O}(mn)$ . The first way to reduce this burden would be to spread out the model update amongst many clients. Indeed, there is no need for every client to send back the model upon every message; the updated model (which is equal across clients) only needs to be sent back once per message. For instance, the client that sent the message could be responsible for sending back the model. Thus, the added network traffic would scale only with the size of the model, i.e.  $\mathcal{O}(m)$ .

Recall that a model is a folded representation of the messages that came before. Thus, to reduce the distribution burden further, we can introduce a “behindness factor”  $b$ , representing how often to ask for a model update. When  $b = 1$ , the server would request the model upon each message being sent (i.e. what was described at the end of the previous paragraph). When  $b = 10$ , the server would request the new model every 10 global messages. This amortizes the cost of serializing and uploading the model across many messages.

If a client joins when the current queue contains, for instance, 8 messages (that is, the current model is called  $G_{n-8}$ ) then the client will be sent  $G_{n-8}$  as well as  $m_{n-7}, \dots, m_n$ , and will be responsible for replaying the 8 messages on top of the given global model. This means that this scheme is also robust to clients leaving. If a client leaves before sending the global model as requested, this will not cause any consistency problems, and another client can later update the server with the global model.

$b$  should be chosen to balance startup time with network traffic. One option is to choose  $b$  to be the ratio of the average size of the model to the message. If that ratio is 100, choosing  $b = 100$  will, on an amortized basis, double the amount of network traffic (every 100 messages, 100 messages’ worth of global model must be uploaded) if

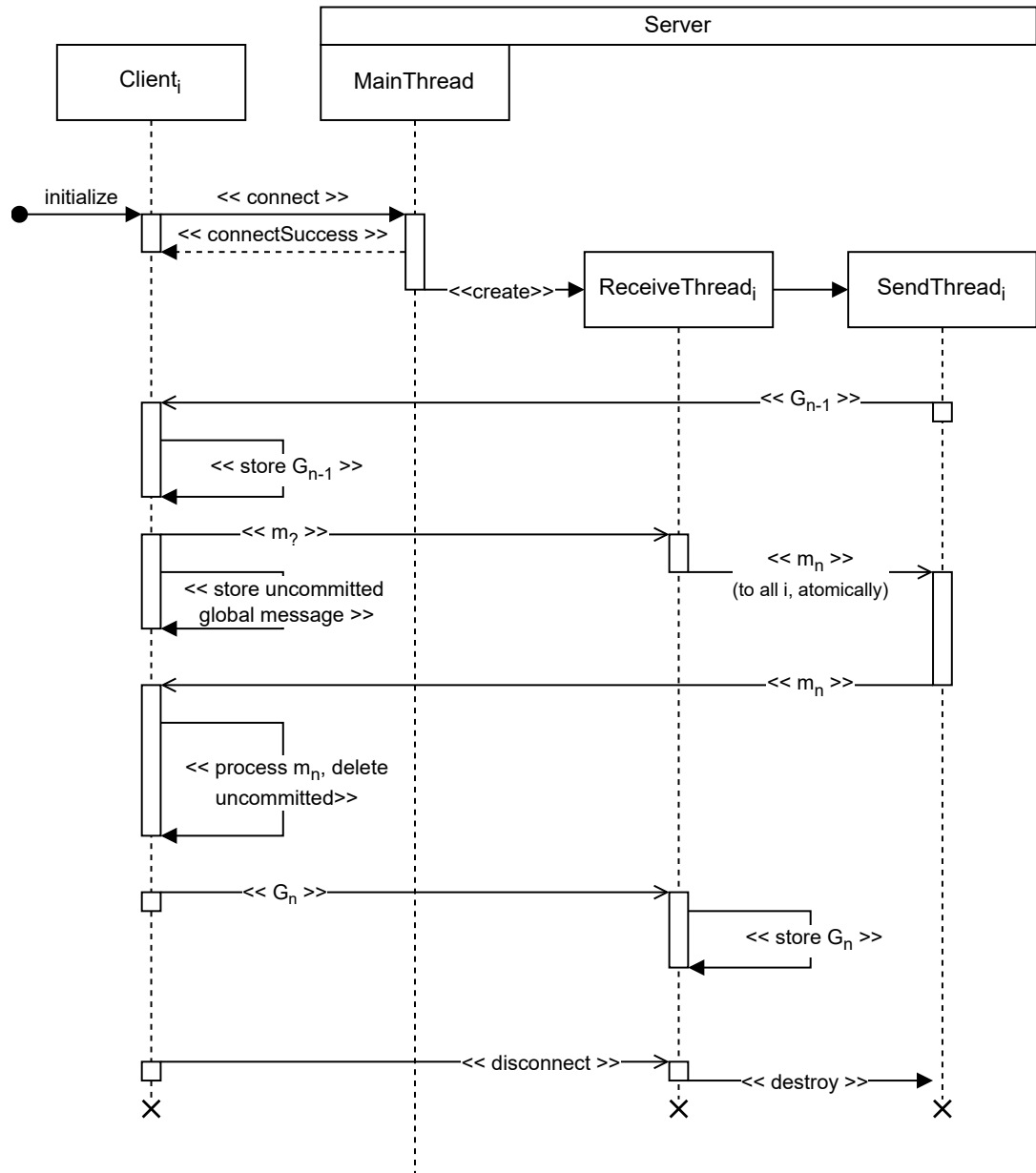


Figure 4.7: Sequence diagram showing the lifecycle of the message-based synchronization scheme with distributed folding. This example has  $b = 1$ . Compared with Figure 4.2, this scheme does not need a headless client running on the server, but instead relies on the clients to send back the global model from time to time.

there was only one client. Adding more clients amortizes the cost across more clients, but each one presumably adds more message burden. We estimate that a value of  $b$  between 10 and 500 would be appropriate for most cases.

Choosing which client does the model upload is also a question. In the simple case, the client that sends the message which fills the queue up to  $b$  will be asked for the model. If all clients send the same number of messages on average, they will share the model upload burden. A client sending more messages would naturally bear a larger share of the burden. Another option would be a pure round-robin, where clients always share the model upload burden equally. However, it may make sense to have the clients who have sent the *least* messages recently update the model, which may increase overall average system responsiveness (clients with lower outbound traffic would be called upon to perform this task). Another option is to take the user's network conditions into account, and give the model upload task to those with the best connection.

## Security Implications

The security properties of this system are not a current topic of study, but would be an interesting topic of research in the future. As with any distributed system with untrusted clients, security is difficult to manage. In all cases, validating the content of the messages being sent would be necessary. In the future, a data access model for TEASync should be developed.

The model-based and distributed folding schemes would be more vulnerable to security problems than other versions, since a malicious client could inject a fake global model. Checksums or consensus-based strategies would help to reduce security

problems.

### **Final Word on Synchronization Schemes**

As mentioned, global message-based synchronization with folding (MsBF) was chosen as the implementation for its simplicity, low network overhead, and lower chance of race conditions. All future discussion in the thesis will be about this scheme.

## **4.5 Desireable Properties of Global Models and Update Functions**

The chosen global message-based synchronization scheme reduces many issues related to concurrency. In fact, in our testing with first-year students, many of which had little-to-no concurrency experience, there was no mention of concurrency issues getting in the way.

Since the server orders the messages and guarantees that each client receives them in the same order, *consistency* amongst clients is guaranteed. As described, message-based synchronization eliminates the possibility of models overwriting each other. However, due to differences in client ping times to the server it is impossible to fully eliminate race conditions caused since messages may arrive at the server in a different order than they were originally sent. However, designing the global update function and messages with properties like associativity (which is automatic), symmetry and idempotency can reduce or eliminate these issues.

Table 4.3: Symmetry Example

Time (s)	Client 1	Client 2	Global Model Value
0	Increment		1
0.5		Decrement	0
1	Decrement	Increment	0
1.5		Increment	1
2		Decrement	0

### 4.5.1 Symmetry

A symmetric relation, also known as a commutative relation, is the property that the ordering of inputs to a relation (in this case, the `globalUpdate` function), does not matter.

#### Definition 4.5.1: Symmetric Global Update

A symmetric global update function is a function `globalUpdate` such that:

$$\forall m, n . m \star n = n \star m$$

For instance, consider the ordering of messages in Table 4.3. At  $t = 1s$ , there is a race condition wherein the server may process the two clients' messages in either order. Luckily, because  $\text{Increment} \star \text{Decrement} = \text{Decrement} \star \text{Increment}$ , *eventual consistency* holds; that is, whether the counter is incremented or decremented first is of no import. In either case, the final value of the global model is `{ count = 0 }`, as expected.

### 4.5.2 Associativity

Associativity is a property of a relation where the insertion of brackets into an operation does not change its output. It is related to the property of symmetry in that it

deals with the `globalUpdate` function being unperturbed by the order of messages.

It is defined mathematically in Definition 4.5.2.

**Definition 4.5.2: Associative Global Update**

A symmetric global update function is a function `globalUpdate` such that:

$$\forall m, n, o . (m \star n) \star o = m \star (n \star o)$$

Since function composition is associative, this property holds automatically.

### 4.5.3 Idempotency

A idempotent relation is one where applying the relation multiple times does not change the value of the model, as described in Definition 4.5.3.

**Definition 4.5.3: Idempotent Global Model**

An idempotent global update function is a function `globalUpdate` such that:

$$\forall m . m \star m = \text{globalUpdate } m$$

Figure 4.8 shows an example adventure game as a state diagram, where states are the locations a player can visit (stored in the global model) and transitions are the pathways the players can take. Consider this TEASync application with two clients. If the current state of the global model is **Bazaar**, and if both clients send a **ToJungle** message at the same time (that is, the second client sends the message before the message arrives), then both clients would receive the **ToJungle** message twice, once when the global model was still **Bazaar** and again when the global model is already **Jungle**. The former case is expected and is encoded in the state diagram, but the latter case is

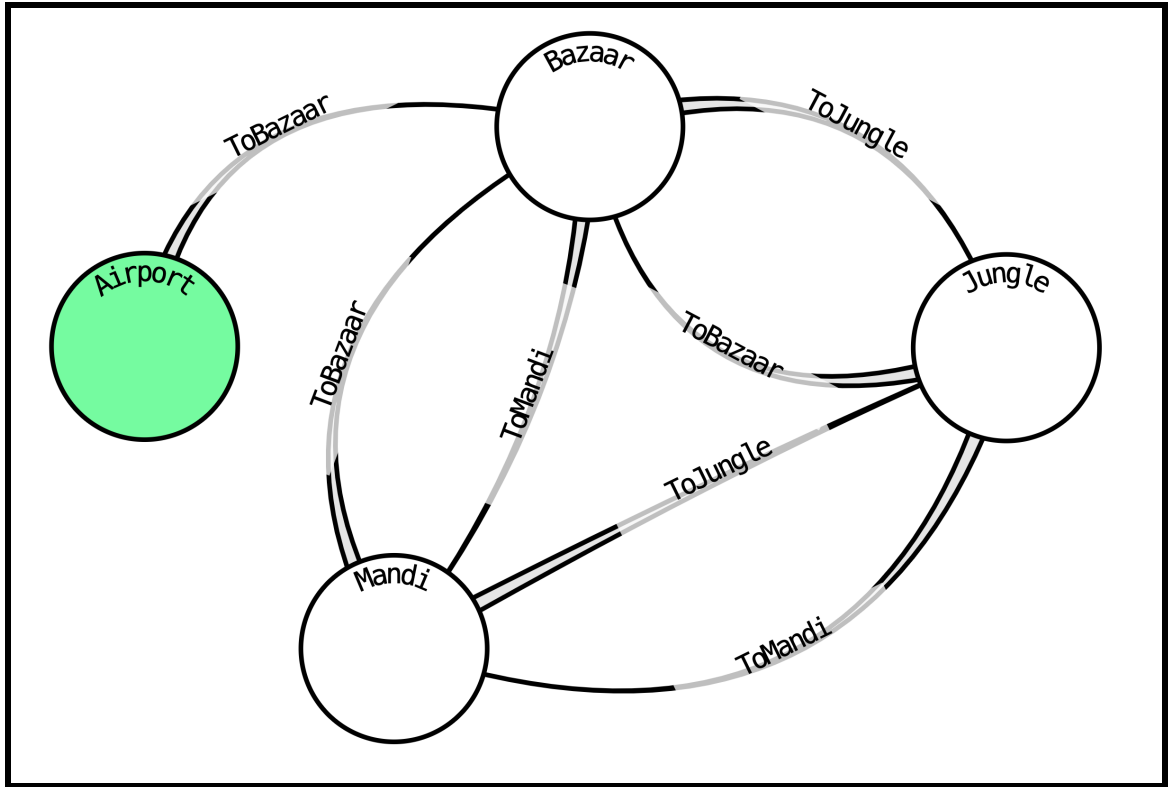


Figure 4.8: Example adventure game shown as a state diagram. States are the locations that players can visit (stored in the global model) and transitions are the pathways a player can take (encoded as global messages).

unexpected and is undefined. However, if we design our `globalUpdate` function properly, the second `ToJungle` message should be “ignored” (i.e. should produce `Jungle` as output). That is,  $\text{ToJungle} \star \text{ToJungle} = \text{globalUpdate ToJungle}$ .

#### 4.5.4 Global Model Design Guidelines

This is both a property of the update function and also a design philosophy for the model, which both go hand-in-hand. This can be broken up into three guidelines:

- G1. Keep global messages as small as possible.



G2. Keep model updates in the `globalUpdate` as small as possible.

G3. Design the global model with orthogonality where appropriate.

The global model-based synchronization model was undesirable in part because even small changes would lead to the entire model being sent to every client, with a loss of semantic information about which parts of the model changed. This led to an extremely high chance of race conditions wiping out other updates. The global message-based approach helps with this, but this by itself does not guarantee desirable properties. For instance, it would be very possible to design a global message type as follows:

```
type GlobalMsg =  
    NewGlobalModel GlobalModel
```

This would effectively simulate global model-based synchronization using global message-based synchronization. Guideline G1 deals with this. Global messages should be designed to be as small as possible, containing all the necessary information for the `globalUpdate` function to operate, but no more. This is usually “obvious” since doing more would add work and complexity, but beginners and veteran programmers alike often over-specify their models and messages, a problem akin to a non-normalized database.

G2 states that functions should minimize the number of reads from and writes to<sup>2</sup> the global model. This helps to reduce the chance of race conditions by ensuring

---

<sup>2</sup>This is an abuse of language since values in Elm are immutable. So, conceptually we do not actually *overwrite* anything in the model. For simplicity, we use “write” to mean a change in the outputted model compared to the inputted one. But internally, Elm’s immutable data structures are implemented in an efficient way, stored as tree types with references from modified data types to their older incarnations. This means that while data structures are conceptually immutable they do not incur the memory copying penalty of being handled that way internally.

the minimal amount of information is changed in an update. One advantage of this architecture is that reads from the global model are cheap, since the data is always cached on the client.

G3 deals in particular with designing models that make use of orthogonality with respect to the different parts of the model. In combination with G2, this helps to reduce or eliminate the possibility of race conditions by ensuring unrelated data is unrelated mathematically. This is achieved by designing a model that makes use of product types when necessary. A word of caution though is that this should not supersede the general guideline to design models that reduce cardinality as much as possible, as previously discussed in Section 3.3.5.

#### **4.5.5 Final Word on Properties**

While the properties are stated using universal quantification to match standard mathematical notation, programs are often more complex than what can be easily captured by formal mathematics. Often, having these properties hold fully is impossible or undesirable.

For instance, we do not always want idempotency to hold for all messages. Indeed, an adventure game with a coin counter (e.g. the game in Figure 4.8 with an embedded version of the counting example) should not be idempotent in all cases; picking up two coins in a row should increment the counter twice. These properties are important guidelines to design programs that avoid strange behaviour, but they must be balanced by the program designer. One should strive to make portions of the program idempotent, symmetric, etc. where it is helpful to do so. In this example, the coins and the current player location would be best encoded as a product data

type. This means that these two features are effectively orthogonal and thus a lack of idempotency on the coin collection does not affect the player location. A more accurate statement of the definitions in this section would therefore take into account orthogonal parts of the model.

While the students were able to make applications without considering these properties, as their programs became more complex, they would be more likely to run into race conditions and thus introducing these would become more and more important.

## Chapter 5

# TEASync Framework Implementation

The purpose of this chapter is to deepen the architectural overview in Chapter 4 by delving into more details about the implementation of TEASync. In particular, the implementation of the Message-Based synchronization scheme with Folding (MsBF). We begin by elaborating on the Application Programming Interface (API) that programmers can use to create their TEASync client in Elm, including the simplified and advanced versions. Next, we discuss the implementation of the codec code generation, including the JSON and binary formats, and how testing was performed using property-based testing techniques. Next, we present the development mode and our online Integrated Development Environment (IDE) for collaborating on, testing, and deploying TEASync applications.

### 5.1 Application Programming Interface

As is standard in the Elm language, TEASync applications are specified by populating a record with pure functions and initial values for models. Defining those values

requires the specification of several associated types. Application developers will often start by defining the types, but the process is usually iterative, with new features requiring both additions to data types and modifications of functions.

### 5.1.1 Simplified API

The simplified API is one without the advanced TEA features subscriptions and commands, and is suitable for beginner applications. The record type utilized is shown in Definition 5.1.1.

#### Definition 5.1.1: TEASync Simple API

The following record type defines a TEASync LG-MVU application. The type is parameterized by the local and global model and message types, and the type of rendering (`viewType`). The developer provides implementations for each of these fields.

```
type alias TEASyncSimpleAppConfig localModel globalModel ↔  
  localMsg globalMsg viewType =  
  { initLocal : localModel  
    , initGlobal : globalModel  
    , localUpdate :  
      localMsg -> localModel -> globalModel  
      -> localModel  
    , globalUpdate :  
      globalMsg -> globalModel  
      -> globalModel  
    , view :
```

```
    localModel -> globalModel
    -> viewType
    , codecGlobalModel : Codec globalModel
    , codecGlobalMsg   : Codec globalMsg
  }
```

where

<code>initLocal</code>	is the initial value of the local model;
<code>initGlobal</code>	is the initial value of the global model;
<code>localUpdate</code>	is the local update function, which is usually defined as a named function and passed in as a first-class value;
<code>globalUpdate</code>	is the global update function, which is usually defined as a named function and passed in as a first-class value;
<code>view</code>	is the view function, which is parametric in its output, allowing any type of rendering (e.g. <code>Html</code> , <code>Svg</code> , etc.);
<code>codecGlobalModel</code>	is the generated codec for the global model, allowing the user to select JSON or binary encoding/decoding;
<code>codecGlobalMsg</code>	is the generated codec for the global messages, allowing the user to select JSON or binary encoding/decoding.

Example B.0.1 in Appendix B shows an example of the simplified API for a collaborative version of the counting example previously shown in Example 3.3.2. The count now becomes part of the global state, with increment and decrement actions in the global message type. The local state keeps track of whether the buttons are being moused over to highlight them. The key part to note is that the entire codebase is

written in Elm on the client, and the server contains no application-specific code.

### 5.1.2 Advanced API

Definition 5.1.2 shows the advanced API for TEASync applications, which includes subscriptions and commands. This can be used to create more advanced applications that require these features. Local commands and subscriptions work similarly to normal MVU programs; that is, they remain local to the client. Global commands and subscriptions are evaluated locally on the client before the resulting message is sent to the server to be shared as usual.

Due to the decentralized nature of TEASync applications, there is no way to have a global subscription (e.g. a timer) emanating from a central client, which unfortunately limits the usefulness of global subscriptions. This can be slightly overcome by designating certain clients as “hosts”, and using the local update function to send relevant global commands on queue.

#### Definition 5.1.2: TEASync Advanced API

The following record type defines a TEASync LG-MVU application. The type is parameterized by the local and global models and messages, and the type of rendering (`viewType`). The developer provides implementations for each of these fields.

```
type alias TEASyncAppConfig localModel globalModel ↔
  localMsg globalMsg viewType =
  { initLocal : (LocalModel, Cmd LocalMsg)
    , initGlobal : (LocalModel, Cmd LocalMsg)
```

```

, localUpdate :
    localMsg -> localModel -> globalModel
    -> ( LocalModel, Cmd LocalMsg, Cmd GlobalMsg )
, globalUpdate :
    globalMsg -> globalModel
    -> ( GlobalModel, Cmd GlobalMsg, Cmd LocalMsg )
, view :
    localModel -> globalModel
    -> { title: String, body : Collage (TEASync.Msg ←
        LocalMsg GlobalMsg GlobalModel) }
, localSubscriptions : LocalModel -> GlobalModel -> ←
    Sub LocalMsg
, globalSubscriptions : LocalModel -> GlobalModel -> ←
    Sub GlobalMsg
, codecGlobalModel : Codec globalModel
, codecGlobalMsg : Codec globalMsg
}

```

where

<code>initLocal</code>	is the initial value of the local model and an initial local command to run;
<code>initGlobal</code>	is the initial value of the global model and an initial global command to run;
<code>localUpdate</code>	is the local update function, which is usually defined as a named function and passed in as a first-class value, which can produce local and global commands;



<code>globalUpdate</code>	is the global update function, which is usually defined as a named function and passed in as a first-class value, which can produce local and global commands;
<code>view</code>	is the view function, which is parametric in its output, allowing any type of rendering (e.g. <code>Html</code> , <code>Svg</code> , etc.);
<code>localSubscriptions</code>	is a function instructing the Elm runtime to send local messages when certain events happen
<code>globalSubscriptions</code>	is a function instructing the Elm runtime to send global messages when certain events happen
<code>codecGlobalModel</code>	is the generated codec for the global model, allowing the user to select JSON or binary encoding/decoding;
<code>codecGlobalMsg</code>	is the generated codec for the global messages, allowing the user to select JSON or binary encoding/decoding.

In regular Elm apps, restricting the use of commands makes sense for a long time, until “advanced” features are needed. In TEASync, commands are often used to communicate between the local and global update functions. Thus, the simplified API was found to quickly become restrictive to groups. A better approach may be to use a hybrid model where the local update function can send global messages directly, and vice versa.

The Pong code in Example B.0.2 in Appendix B shows how commands can be used to handle more complex cases like physics. The physics rendering is done on the client (in the local messages/model), while the global messages are used to synchronize at

key points in the simulation; namely, when the ball hits a player's paddle. Commands are used to send messages from the local update to the global update in this case using a function called `newMsg`. This illustrates a drawback of TEASync's distributed design as there is no centralized time tick to allow for global simulations. This approach has the downside of being more complicated but has clear upsides in terms of network traffic and the smoothness of the simulation.

## 5.2 Encoder/Decoder (Codec) Generation

This section discusses the implementation of the codec generation, including tokenization, parsing, a representation of Elm's Hindley-Milner type system in Haskell, and then how JSON and binary codec generation works.

### 5.2.1 Tokenization

The tokenization scheme for Elm types is shown in 5.2.1.

#### Definition 5.2.1: Tokens for Elm Types

Below are the tokens for the subset of the Elm language that allows parsing types with comments interspersed:

```
<type> ::= "type"
<alias> ::= "alias"
<unit> ::= "()"
<eq> ::= "="
<pipe> ::= "|"
```

```
<type> ::= "type"

<lbrack> ::= "{"
<rbrack> ::= "}"
<lparen> ::= "("
<rparen> ::= ")"
<comma> ::= ","
<period> ::= "."
<colon> ::= ":"
<arrow> ::= "->"
<newline> ::= "\n"

<comment> ::= <sl-comment> | <ml-comment>

<sl-comment> ::= "--" { <text> } <newline>
<ml-comment> ::= "{-" { <text> } "-}"

<block-end> ::= ^<non-space>a

<other-token> ::= *b
```

<sup>a</sup> We abuse the notation of EBNF here. This token is needed to know when a definition ends and another begins. Elm, like Haskell, uses the very left-hand column to denote new blocks. As long as a line starts with a space, the tokens are part of the same block (e.g. the same type). If a line starts with a non-space character, we insert a **<block-end>** token so the parser has that information. We use **^** to represent the start of a line, as in regular expressions, and **<non-space>** represents any non-space character.

<sup>b</sup> We skip the rest of the Elm grammar, as we are only interested in type definitions.

## 5.2.2 Parsing

Definition 5.2.2 shows the parser used for Elm types, assuming the stream has already been converted from text into a sequence of tokens.

### Definition 5.2.2: Parser for Elm Types

Below is the EBNF scheme for parsing Elm types. This uses non-terminals from Definitions 5.2.1. The parser assumes that the raw text has already been tokenized.

```

<type-decl> ::= (<union-type> | <type-alias>) <block-end>

<union-type> ::= <type> <upper-var> { <type-var> } <eq>
                <constr> { <pipe> <constr> }

<type-var> ::= <lower-var>

<constr> ::= <upper-name> { <simple-type> }

<simple-type> ::= <base-type> | <type-var> | <record> | <unit>
                | <tuple> | <type-name>
                | <lparen> <type-app> <rparen>

<base-type> ::= <bool> | <int> | <string> | <float>

<record> ::= <lbrack> { [ <rec-field> { <comma>
                <rec-field> } ] } <rbrack>

<tuple> ::= <lparen> <smpl-or-app> [<tuple-field>
                [<tuple-field>]] <rparen>

<tuple-field> ::= <comma> <smpl-or-app>

```

```
<smpl-or-app> ::= <simple-type> | <type-app>

<rec-field> ::= <lower-var> <colon> <smpl-or-app>

<type-name> ::= <upper-name> { <period> <upper-name> }

<type-app> ::= <type-name> { <type-name> }

<type-alias> ::= <type> <alias> { <type-var> } <eq> <smpl-or-app>
```

### 5.2.3 Elm Hindley-Milner Type Representation

We represent Elm’s Hindley-Milner types using the type defined in Definition 5.2.3

#### Definition 5.2.3: Haskell Type for Elm Types

Below is the data type used to store Elm types in the TEASync Haskell code, after being parsed from the user’s Elm types module. These are used to generate the encoders and decoders needed to send and receive the user’s data (including their global model and messages).

```
data TypeDecl
    = TUnion Name [(Name, [Type])]
    | TAlias Name Type
    deriving (Eq, Show)

type Name = Text

data BaseType
```

```
    = TBool
    | TInt
    | TString
    | TFloat
    deriving (Eq, Show)

data Type
    = TUnit
    | TVar Name
    | TBaseType BaseType
    | TLambda Type Type
    | TTypeApp [Name] Name [Type]
    | TRecord [(Name, Type)] (Maybe Name)
    | TTuple Type Type (Maybe Type)
    deriving (Eq, Show)
```

#### Definition 5.2.4: Find Type Variables Function

Below is the definition of the `findTypeVars` function, defined recursively by pattern-matching. This function recursively finds all type variables in a given type. For simplicity, we abuse the notation a bit by allowing the function to be named the same name despite taking in two different types.

$ty :=$	typevars $ty :=$
TUnion $_$ $constrs$	$\bigcup (c, ts) \in constrs . \bigcup t' \in ts . \text{typevars } t'$
TAlias $_$ $t$	typevars $t$
$ty :=$	typevars $ty :=$
TUnit	$\{\}$
TVar $n$	$\{n\}$
TBaseType $bt$	$\{\}$
TLambda $a$ $b$	typevars $a \cup \text{typevars } b$
TTypeApp $_$ $_$ $ts$	$\bigcup t \in ts . \text{typevars } t$
TRecord $fs$ $_$	$\bigcup (f, t) \in fs . \text{typevars } t$
TTuple $a$ $b$ Nothing	typevars $a \cup \text{typevars } b$
TTuple $a$ $b$ (Just $c$ )	typevars $a \cup \text{typevars } b \cup \text{typevars } c$

### 5.2.4 JSON Codec Generation

Tables 5.1, 5.2 and 5.3 show the definition of the JSON format used to encode/decode the programmer’s data types. The JSON format is described by a conceptual function  $\text{json}_t$ , wherein we slightly abuse the function notation by using the same name despite having different input types. In this way, it forms a sort of class of functions distinguished by the input type, which could be implemented in Haskell using typeclasses. The subscript  $t$  is used to denote the “input” type.

In Table 5.1, we use lambda notation to show that in cases where we have type

Table 5.1: JSON Format for the `TypeDecl` Type

$ty :=$	<code>TUnion name [(c<sub>1</sub>, [t<sub>1,1</sub>, ..., t<sub>1,n<sub>1</sub>])], ..., (c<sub>m</sub>, [t<sub>m,1</sub>, ..., t<sub>m,n<sub>m</sub>])]</sub></sub></code>
<code>json ty :=</code>	<code>λtypevars ty → {"tag":c<sub>i</sub>, "f1":json t<sub>i,1</sub>, ..., "fn":json t<sub>i,n</sub>}</code>
$ty :=$	<code>TAlias name t</code>
<code>json ty :=</code>	<code>λtypevars t → json t</code>

Table 5.2: JSON Format for the `BaseType` Type

$ty :=$	<code>json ty :=</code>
<code>TBool</code>	<code>true/false</code>
<code>TInt</code>	<code>0, 1, 42, etc.</code>
<code>TString</code>	<code>"Hello World"</code>
<code>TFloat</code>	<code>2.7182818</code>

variables, these become inputs to the `jsont` function. In Table 5.3 these are passed in as “arguments” in the `TTypeApp` case. Table 5.9 illustrates some examples of generated encoders and decoders and example JSON code to make this a bit clearer.

### JSON Format for `TypeDecl`

Table 5.1 shows the JSON format for union type declarations and type aliases, which recursively use formats for other kinds of types.

### JSON Format for `BaseType`

The JSON format of Elm base types is given in Table 5.2.

### JSON Format for `Type`

The JSON format of Elm types is given in Table 5.3.



Table 5.3: JSON Format for the `Type` Type

$ty :=$	<code>json ty :=</code>
<code>TUnit</code>	<code>&lt;none&gt;</code>
<code>TVar n</code>	<code>json n</code>
<code>TBaseType bt</code>	<code>json bt</code>
<code>TTypeApp "List" m [t]</code>	<code>[ json t, ..., json t ]</code>
<code>TTypeApp _ m [t<sub>1</sub>...t<sub>n</sub>]</code>	<code>json m json t<sub>1</sub> ... json t<sub>n</sub></code>
<code>TRecord [(f<sub>1</sub>, t<sub>1</sub>)... (f<sub>n</sub>, t<sub>n</sub>)] _</code>	<code>{"f1":json t<sub>1</sub>, ..., "tn":json t<sub>n</sub>}</code>
<code>TTuple a b Nothing</code>	<code>{"fst":json a, "snd":json b}</code>
<code>TTuple a b (Just c)</code>	<code>{"fst":json a, "snd":json b, "thd":json c}</code>

Table 5.4: JSON Format for Select Standard Library Types

$ty :=$	<code>json ty :=</code>
<code>Maybe a</code>	<code>json (TUnion "Maybe" [("Just", [TVar "a"]), ("Nothing", [])])</code>
<code>Set a</code>	<code>json (TTypeApp "List" [TVar "a"])</code>
<code>Dict k v</code>	<code>json (TTuple (TVar "k") (TVar "v") Nothing)</code>

### JSON Format for Select Standard Library Types

The JSON format of selected standard library types is given in Table 5.4. These include the `Maybe` and `Set` types, and the `Dict` types. They are implemented as helper functions that are built upon the `json` functions already specified (i.e. union types, lists, and tuples).

#### 5.2.5 Binary Codec Generation

TEASync also has a binary format for data types, which should allow for much smaller encodings most of the time, at the cost of not being easily human-readable. This subsection discusses the design of this format. The design of this was inspired by Elm creator Evan Czaplicki’s blog post detailing a future binary format for Elm types [12], but modified to be more compact. The purpose of the binary format developed by

Table 5.5: Binary Format for the `TypeDecl` Type

$ty$	$:=$	<code>TUnion</code>	<code>name</code>	$[(c_1, [t_{1,1}, \dots, t_{1,n_1}]), \dots, (c_m, [t_{m,1}, \dots, t_{m,n_m}])]$
<code>bin</code>	$ty$	$:=$	$\lambda \text{typevars } ty \rightarrow$	<code>tag</code> $c_i$ <code>&lt;&gt;</code> <code>bin</code> $t_{i,1}$ <code>&lt;&gt;</code> $\dots$ <code>&lt;&gt;</code> <code>bin</code> $t_{i,n_i}$
$ty$	$:=$	<code>TAlias</code>	<code>name</code>	$t$
<code>bin</code>	$ty$	$:=$	$\lambda \text{typevars } t \rightarrow$	<code>bin</code> $t$

Czaplicki was to eventually use the format to replace the internal representation of Elm’s data types. As such, it was important to have constant-time lookup of any field in the data. Since we are currently limited to always decoding the values, which is a linear process anyway, we have optimized away the need for certain information, like lengths, in TEASync’s binary format design.

As before, the `bint` notation will be used to represent the binary format.

## Binary Format for `TypeDecl`

Table 5.5 shows the binary format for union type declarations and type aliases, which recursively uses formats for other kinds of types. For union types, the constructor is identified first, followed by any data fields in that constructor. This is shown by the `tagci` function, which assigns a unique number to each constructor. To save space, we adjust the number of bits needed by taking the base-2 logarithm of the number of constructors. For instance, data types with up to 256 constructors only need an 8-bit (single-byte) unsigned integer to uniquely identify them. Most of the sum types created by programmers in well-written programs should fall into this category, but code-generated programs may contain types with many more constructors. The maximum number of constructors supported is  $2^{32}$ , which uses a 4-byte unsigned integer to represent the constructor.

Table 5.6: Binary Format for the `BaseType` Type

$t :=$	Value	$\text{bin}_t$	Size (Bytes)
TBool	true/false	01/00	1
TInt	1, 42, etc.	00 00 00 01, 00 00 00 2A, etc.	4
TString	"Hello"	00 00 00 05 48 65 6C 6C 6F	$4 + \text{len}(\text{str})$
TFloat	2.7182818	40 2D F8 54	8

### Binary Format for `BaseType`

The binary format of Elm base types is given in Table 5.6. Booleans use an entire byte to represent them, which is obviously wasteful. This is due to Elm’s `bytes` package being limited to full bytes. Implementing byte packing would help to solve this, but the added complexity of the code generation was deemed not worth it. Strings are tagged with a 32-bit unsigned integer representing their length, allowing strings of up to length  $2^{32}$ , which at 4gb is an extremely generous upper limit. Floating-point numbers use 64-bit representations by default.

### Binary Format for `Type`

The binary format of Elm types is given in Table 5.7. As with the JSON format, most of these are their obvious recursive definitions. Instead of needing brackets and commas to delimit values, the bytes are simply appended to one another and their lengths are recursively inferred (when possible) or explicitly stated if needed (e.g. strings, lists).

### Binary Format for Select Standard Library Types

The binary format of selected standard library types is given in Table 5.8. These include the `Maybe` and `Set` types, and the `Dict` types. They are implemented as

Table 5.7: Binary Format for the Type Type

$ty :=$	$\text{bin } ty$	Size (Bytes)
<b>TUnit</b>	<code>&lt;none&gt;</code>	0
<b>TVar</b> $n$	$\text{bin}_n$	$ \text{bin}_n $
<b>TBaseType</b> $bt$	$\text{bin}_{bt}$	See Table 5.6
<b>TTypeApp</b> "List" $m$ $[t]$	$\text{bin } t <> \dots <> \text{bin } t$	$\sum_i^n  \text{bin } t_i $
<b>TTypeApp</b> $_m$ $[t_1 \dots t_n]$	$\text{bin } m \text{ bin } t_1 \dots \text{bin } t_n$	$ \text{bin } m \text{ bin } t_1 \dots \text{bin } t_n $
<b>TRecord</b> $[(f_1, t_1) \dots (f_n, t_n)]$ $_$	$\text{bin } t_1 <> \dots <> \text{bin } t_n$	$\sum_i^n  \text{bin } t_i $
<b>TTuple</b> $a$ $b$ <b>Nothing</b>	$\text{bin } a <> \text{bin } b$	$ \text{bin } a  +  \text{bin } b $
<b>TTuple</b> $a$ $b$ <b>(Just</b> $c)$	$\text{bin } a <> \text{bin } b <> \text{bin } c$	$ \text{bin } a  +  \text{bin } b  +  \text{bin } c $

Table 5.8: Binary Format for Select Standard Library Types

$ty :=$	$\text{bin } ty$
<b>Maybe</b> $a$	<code>json TUnion "Maybe" [("Just",[TVar "a"]), ("Nothing",[])]</code>
<b>Set</b> $a$	<code>json TTypeApp "List" [TVar "a"]</code>
<b>Dict</b> $k$ $v$	<code>json TTypeApp "List" [TTuple (TVar "k") (TVar "v") Nothing]</code>

helper functions that are built upon the `json` functions already specified (i.e. union types, lists, and tuples).

### 5.2.6 Examples and Size Comparison

Table 5.9 shows some example datatypes, with their JSON and binary encodings and the corresponding sizes in bytes (not including spaces). In most cases the binary encoding is much more space-efficient, largely due to the lack of delimiting syntax and field names. This is especially true of union types. While additional work could make the JSON format more efficient, it is intended to be human-readable for debugging purposes. Even so, the binary format would often end up more compact.

There are cases where the JSON format is actually more compact, especially for small numbers that have fewer digits than their binary encoding’s number of bytes. In the future, the TEASync library should provide `Int8`, `UInt16`, etc. data types for

programmers to use when numbers are known to be small. These would signal to the code generator that it can use the more efficient encodings rather than the default ones.

#### Example 5.2.1: Example Datatypes

The following are some example data types. Table 5.9 shows some example JSON and binary encodings from these types, along with their resulting sizes in bytes and their compression ratios. Compression ratios of less than 1 indicate an increase in data size for that example.

```
type TrafficLight =  
    Red | Yellow | Green  
  
type GlobalModel =  
    { count : Int }  
  
type IntList  
    = Empty  
    | Cons Int IntList  
  
type Tree a  
    = Leaf a  
    | Branch (Tree a) a (Tree a)
```

Table 5.9: Example Datatype Encodings and Compression Ratios

Elm Datatype	Example Value	JSON/Hex Encoding	Size (Bytes)
Int	25565	25565 00 00 63 DD	5 4 (1.25x)
List Int	[1, 2, 3, 4]	[1, 2, 3, 4] 04 00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 04	9 17 (0.53x)
TrafficLight	Yellow	{ "tag" : "Yellow" } 01	13 1 (13x)
IntList	Cons 1959 (Cons 1966 Empty )	{ "tag" : "Cons" , "f1" : 1959 , "f2" : { "tag" : "Cons" , "f1" : 1966 , "f2" : { "tag": "Empty" } } } 01 00 00 07 A7 01 00 00 07 AE 00	75 11 (6.8x)
GlobalModel	{ count = 2002 }	{ "count" : 2002 } 00 00 07 D2	14 4 (3.5x)
Tree Int	Branch (Leaf 1887) 1955 (Leaf 1967)	{ "tag": "Branch" , "f1": { "tag": "Leaf" , "f1": 1887 } , "f2": 1955 , "f3": { "tag": "Leaf" , "f1": 1967 } } 01 00 00 00 07 5F 00 00 07 A3 00 00 00 07 AF	86 15 (5.7x)

### 5.2.7 Codec Generation Conclusions

By using Elm’s strong type system, the TEASync library parses the programmer’s `Types` and then generates code to encode and decode values, including the global model and messages, any custom types the user creates, and common built-in and standard library types. This is a model-driven development approach, where the model is in the form of the types in the programmer’s codebase. If they change, the code can easily be regenerated to support the new/updated types. Encoding and decoding functions are mechanical and do not offer much pedagogical benefit beyond being a programming exercise the first few times. By taking care of this aspect, the TEASync framework allows the programmer to focus more on the program design itself.

TEASync supports both a human-readable JSON format and a more compact binary format, which can be used in different circumstances (i.e. development vs. deployment). The binary format often leads to a much more compact encoding, at the cost of much lower human readability.

## 5.3 Module Hierarchy

This section shows the module hierarchy for the TEASync client and server. The client contains many modules that are either static or generated, and users do not have to change many of them. The server is static and does not change depending on the application, thus allowing any number of different TEASync servers to be hosted from a given server instance.

## Client

The module hierarchy of the client is shown in Figure 5.1. The programmer writes their code in the **Main** and **Types** modules, which contain the application code and the types, respectively. The **Types** module is the one which is scanned to generate the encoders and decoders, which are stored in the modules in the **Codec** directory.

The **Main** module is where the programmer fills in their application code as shown in 5.1.1. This includes their **view** function, **init** functions, and **update** functions. They can choose to break out pieces of these into other modules if they would like.

The code inside the **TEASync** directory is the library code for the framework, which can be thought of as the runtime for the TEASync framework. This includes helper functions for encoding and decoding, as well as the headless client that runs on the server in *message-based synchronization with folding*. This will be broken out into a library in the future, but was kept inline for ease of testing and iteration thus far.



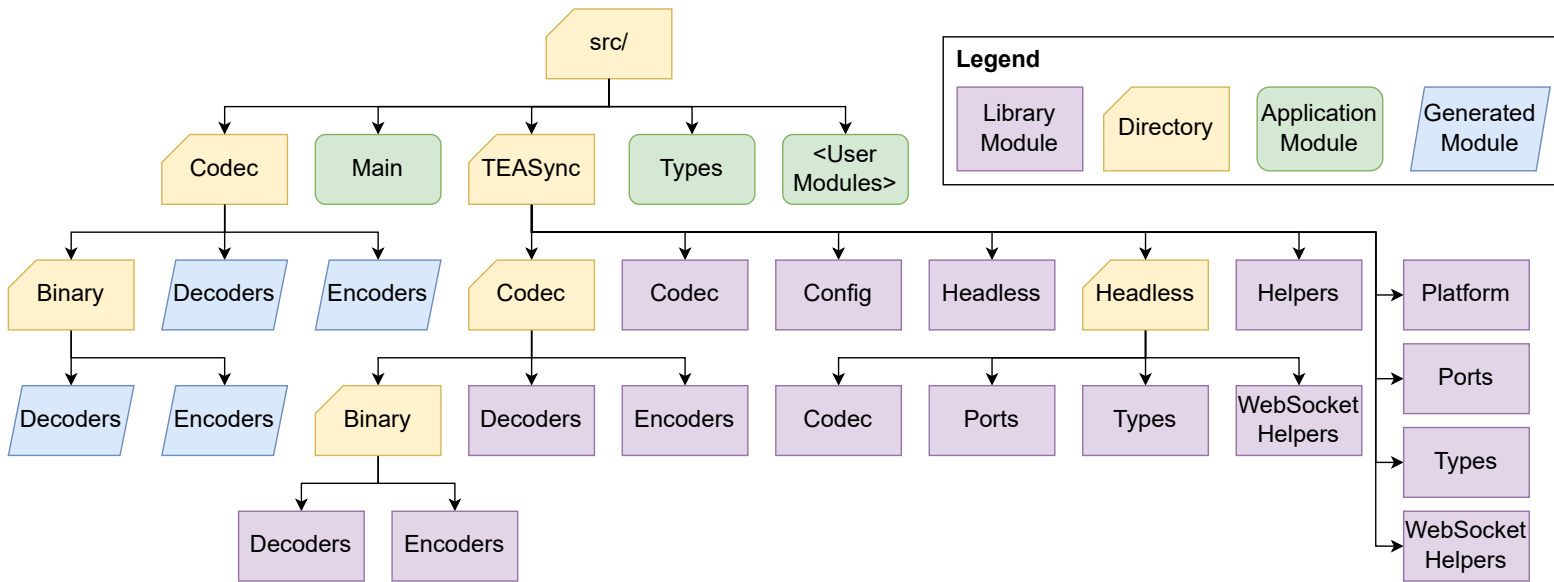


Figure 5.1: The module hierarchy of the TEASync client, showing TEASync library modules, directories, application modules written by programmers, and generated modules. Developers are free to add modules and directories to split up their code into multiple modules. These will be imported by the **Main** module to be included in their final application.

## Server

The module hierarchy of the server is shown in Figure 5.2. It contains modules for lexing and parsing the Elm **Types** module (**Lexer** and **Parser**), generating code (the **Generate** directory), and the data types to make the framework work. The **Web** directory contains the model-view-controller IHP code which gives the programmers a web interface to create and manage their servers.

## 5.4 Development Mode

The advantage of the LG-MVU model is that it does not require a server to run; it can be equally simulated from within a single client. Thus, TEASync includes a development or “offline” mode, allowing the programmer to spawn any number of virtual clients which interact as if it were running on a server. This allows the programmer to test their application without needing to compile and run an entire server.

Figure 5.3 shows the interface with 3 clients running simultaneously, playing a multiplayer game of pong. The TEASync application code contains the logic for running a virtual server, orchestrating the virtual clients and sending/receiving local messages (specific to each virtual client) and global messages (which are mapped over all virtual clients to update their global state). The nature of the LG-MVU model and Elm’s pureness guarantee makes this simulation both simple to implement and safe, since state cannot escape the scope of the **update** and **view** functions.

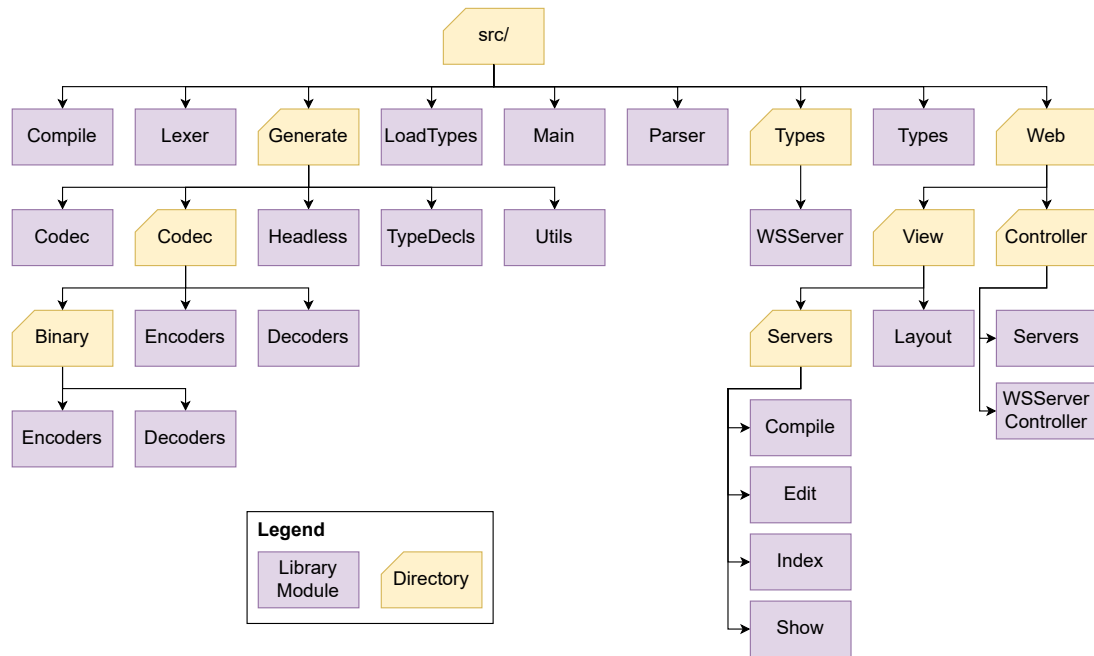


Figure 5.2: The module hierarchy of the TEASync server, showing TEASync library modules and directories. Some modules (especially IHP-specific ones) are omitted for clarity and brevity.

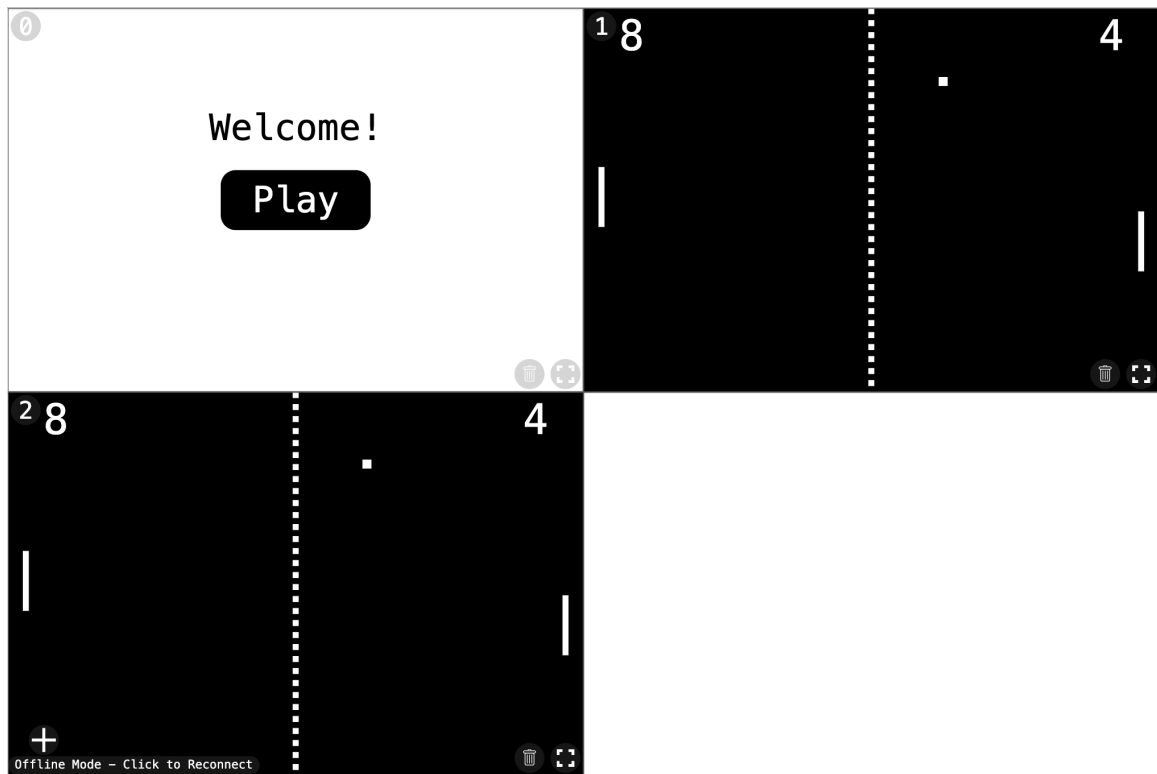


Figure 5.3: A pong game running in development mode. In development mode, programmers can simulate their application by spawning any number of virtual clients (in this case three), which interact like they would when running with a real server. This mode is available to programmers with no additional coding on their part.

## 5.5 Online Collaborative Integrated Development Environment

To allow programmers to easily use the framework, it was important to give them an integrated development and deployment environment that not only allows them to collaboratively develop their application but also allows them to deploy and test their applications easily online. Thus, we extended our `STaBL.Rocks` project system (introduced in Section 5.5.1) with TEASync support. The two innovations that made this possible were the collaborative project system and the TEASync deployment system, which are described here.

### 5.5.1 Collaborative Project System

As introduced previously, the collaborative project system is an extension of `STaBL.Rocks` that allows programmers or teams of programmers to work on projects containing multiple modules.

Modules inside a given project can import each other, as one would expect from a project with multiple modules. However, collaborating on the module level can have drawbacks when multiple people are editing at once. We want something that will help protect teams against bugs, but with much lower complexity than a system like Git. To reduce the chance of bugs in one programmer’s code affecting other modules, we use a *release* system (see Figure 5.4). Releases are accompanied by a descriptive *release message* and can be thought of as per-module commit messages, signalling that a change has been made to a module. A release can only be made if the module compiles successfully. When importing Module A into Module B, the

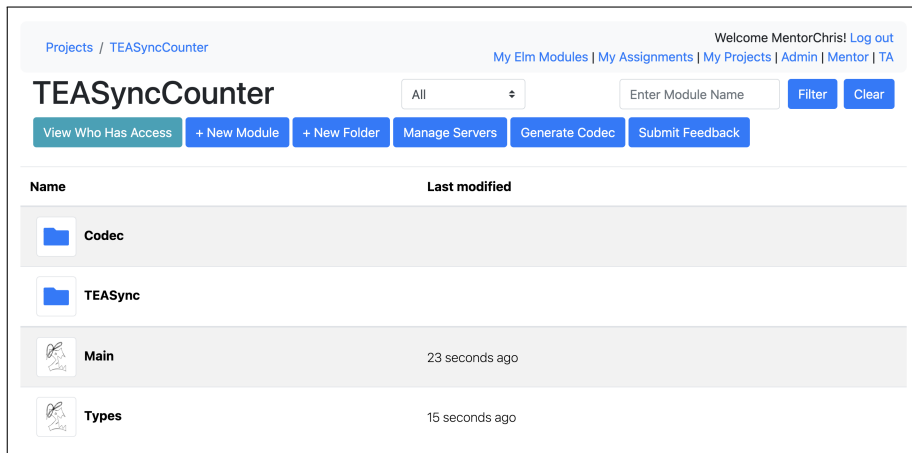
version of Module A used as part of the compilation will be the last-released version of Module A. While this does not eliminate the possibility of breaking API changes in Module A compared to what Module B expects, it does eliminate Module B breaking when Module A is in active development.

Although it is out of scope for this thesis, we are interested in developing a simplified version control system for new programmers more powerful than the present system, but still much easier to learn than professional systems. We are currently investigating the problems people run into with current version control systems, to better understand the problem. The goal is not to replace industry-standard version control tools but rather determine if there are simpler abstractions that help new programmers, even grade school students, learn the basics of version control and why it is important. The release system is a first attempt at this goal and will be polished and improved over time.

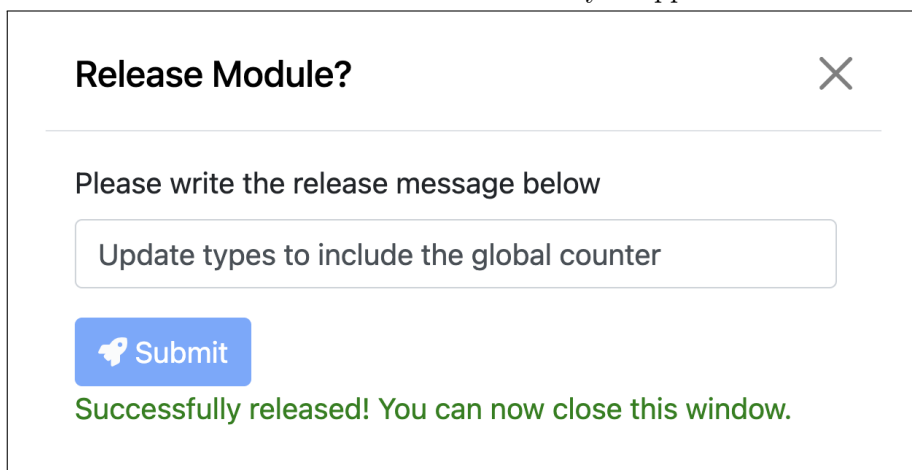
### 5.5.2 Deployment

The next feature of the STaBL.Rocks IDE is the ability to deploy servers live, allowing programmers to quickly perform real-world tests of their applications beyond the simulated development mode.

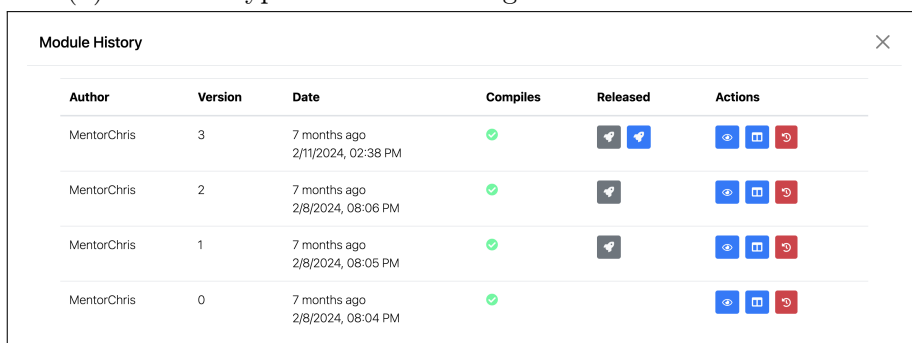
As shown in Figure 5.4a, TEASync projects have the option to manage servers. Clicking this button brings up the instance list screen shown in Figure 5.5. This screen allows the programmer to compile and deploy their TEASync application, providing them a link that can be shared to connect to that instance of their server. Each application can have any number of servers, each with a unique ID and value for the global model. For instance, a two-player game may have many *lobbies* (unique



(a) A TEASync project in the STaBL.Rocks project system, showing the modules and folder of the TEASync application.

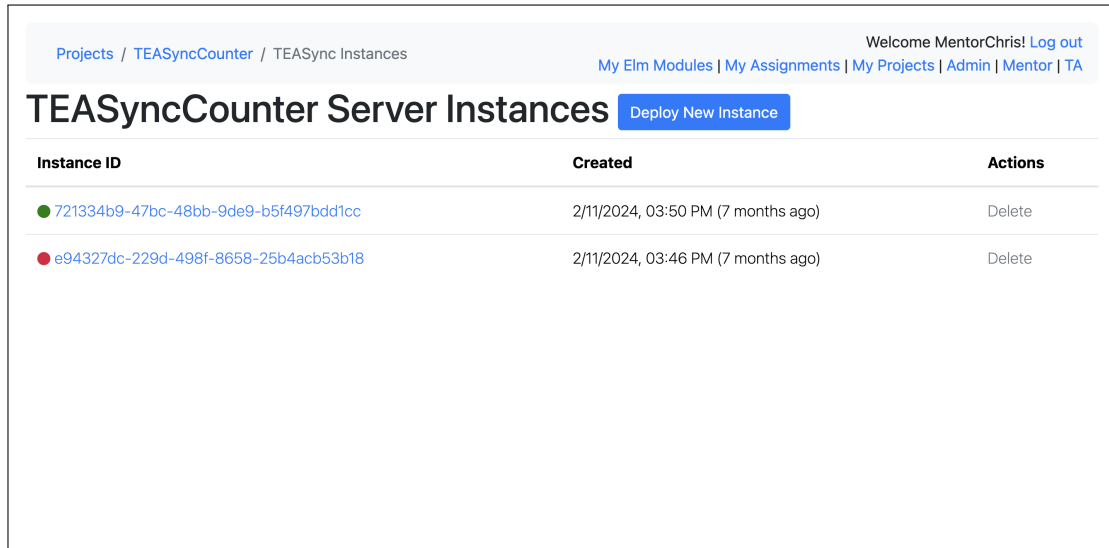


(b) The user types a release message and submits the release.



(c) Now the release shows up in the module history screen, shown as a rocket ship icon.

Figure 5.4: The collaborative project system in STaBL.Rocks is used to create TEASync projects.



Instance ID	Created	Actions
721334b9-47bc-48bb-9de9-b5f497bdd1cc	2/11/2024, 03:50 PM (7 months ago)	Delete
e94327dc-229d-498f-8658-25b4acb53b18	2/11/2024, 03:46 PM (7 months ago)	Delete

Figure 5.5: Each TEASync project can have many server instances, each with a unique identifier and value for the global model. Developers can share a link to access a certain instance.

instances of the game) running at once. Many new programmers will be familiar with this use of the word “server” from experience creating their own Discord “server”.

If the programmer clicks on a certain instance, they can see the server dashboard shown in Figure 5.6. This dashboard allows you to start or stop the server, open the client link to use the application, view the current value of the global model in JSON format, and reset the model back to default, which will restart the server instance in a fresh state.

## 5.6 Implementation using Functional Programming

Elm made for a very natural implementation of the LG-MVU model and the TEASync client framework. While it would have been possible to implement LG-MVU in any language, Elm provides the advantages of type safety and function pureness. This



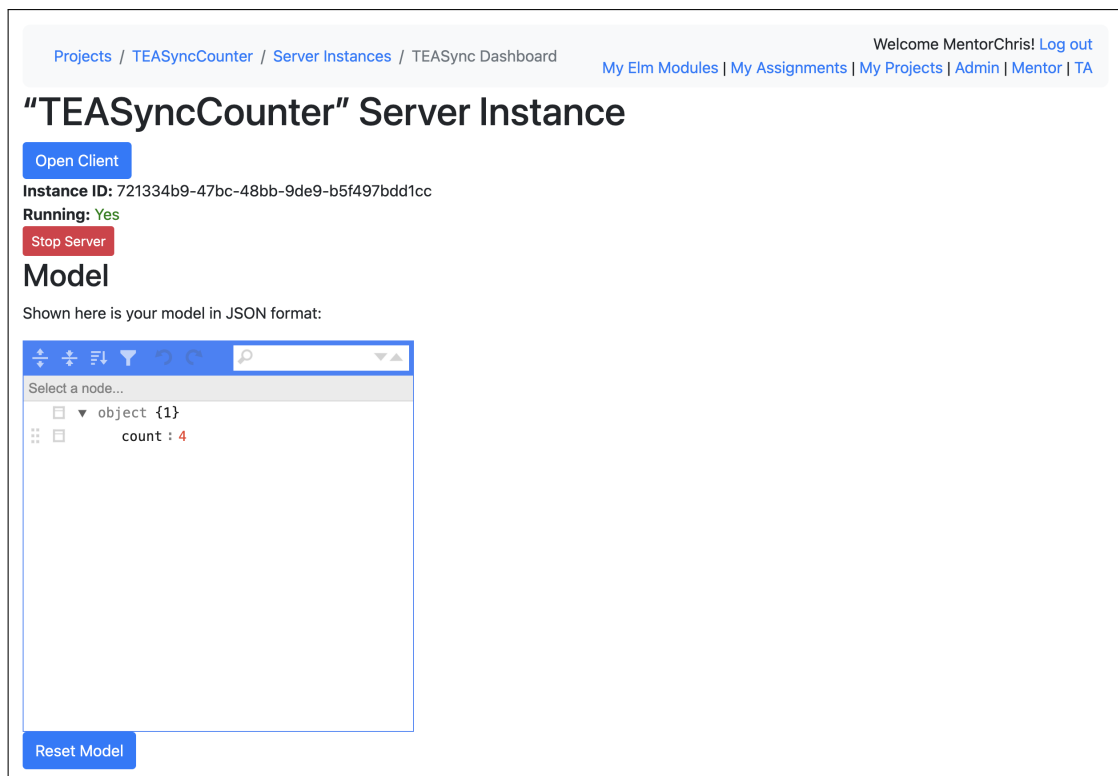


Figure 5.6: The dashboard for server instances allows the developer to start and stop the instance, open the client link, view the current state of the global model in JSON format, or reset the model back to the initial state.

ensures that no state is “leaking in” from outside the functions, making the framework extremely predictable and reliable.

On the server side, Haskell’s lightweight threading system and strong typing were also great assets. They allowed us to have a massively concurrent framework with very little overhead. Software transactional memory also made inter-thread communication reliable and easy to reason about, as well as allowing us to ensure ordering guarantees.

On the front end, it would be possible to use languages other than Elm, and still connect it to the Haskell backend, but non-pure functions open the door to bugs that do not manifest at their source, but somewhere else in the codebase. Given the large number of support functions (including the generated codecs, communication layer and speculative local application of messages) the possibilities for induced errors difficult for beginners to debug would be significant. Furthermore, features like the virtual-client development mode were much easier due to Elm’s pure functions and TEA’s messages which are pure data values.

# Chapter 6

## Usability Study Methodology

This chapter discusses the design of the usability study that was conducted to gather feedback on the TEASync framework. Section 6.1 provides an overview of the study design and describes the goals of the study. This is followed by the proposition of the study in Section 6.2. Next, we describe the design of the surveys in Section 6.3 followed by that of the focus groups in Section 6.4 and the compilation statistics in Section 6.5. Finally, we discuss the threats to validity of the study.

This chapter contains some representative questions as figures. The full surveys and focus group questions can be found in Appendices A.1, A.2, and A.3.

The results of the study are presented in Chapter 7, and conclusions and analysis of those results in Chapter 8.

### 6.1 Overview and Goals

The study was run in a class of 197 first-year computer science students. Most of the students had learned Elm (including our GraphicSVG graphics framework) in a

course the semester prior, but had little experience creating interactive applications using Elm.

Course evaluation consisted of three midterms, a design project, and a take-home design exam. In Midterm 1, students were tasked with adding new features to the code generated from a state diagram (i.e., the template code for a single-player adventure game). Midterm 2 tested knowledge of recursion by asking students to reproduce prescribed recursive pictures using Elm code. The third midterm tested knowledge of TEASync.

Students had two-hour, bi-weekly labs in which they would learn content (such as the state diagram tool, TEASync, etc) or be given time to work on their projects. In the projects, students were tasked with using the Design Thinking (DT) process described in Anand et al. [3]. Their problem brief was to use DT to create a game that would be appealing to older adults, with the goal of detecting Parkinson’s Disease symptoms. Actual measurement of symptoms was outside the scope of the project, but it was necessary to have a story justifying the suitability of the game actions for this purpose. Students were given the option of either creating a single-player game or a multiplayer game using TEASync.

The study, approved by McMaster’s Research Ethics board under project #6868, consisted of four parts:

1. A pre-implementation survey to get a baseline understanding of the students’ backgrounds in computer science and creating games, and their preferences in creating games.
2. A focus group near the end of the course to gather feedback about the framework and the course in general.

3. A post-implementation survey focussed on their experience in the course and using the framework.
4. Reporting of statistics from the STaBL.Rocks platform.

The two surveys and full focus group questions are included in Appendices A.1, A.2, and A.3. Students were invited to participate in any or all parts of the study; no parts of the study required participating in other parts. With the exception of the focus groups, where attendance was not taken, for each part of the study they participated in, students were given an entry in a draw to win a \$20 gift card of the winner's choice.

## 6.2 Proposition

We postulate that new programmers will be able to use Event-Driven Programming to create a multi-user application with the help of tool support. Furthermore, it is expected that those who make a multiplayer game show a measurably increased engagement in the course compared to students who make a single-player game. As a heuristic to measure engagement, we will use the number of interactions with the online Integrated Development Environment (IDE).

## 6.3 Survey Design

The study consisted of two surveys, one given near the beginning of the course and one given in the last week of the course.


### 6.3.1 Pre-Implementation Survey

The Pre-Implementation survey questions were focussed on ascertaining students' backgrounds in terms of their experience with computer science and programming. Combined with their eventual decisions to use the framework or not, this will help us to understand which kinds of students choose to use the framework and why.

The Pre-Implementation survey's questions are shown in Appendix A.1. Figure A.1 shows the informed consent questions. Section 2 contains questions about their past experience with computer science and game development (examples in Figure 6.1). The full Section 2 is shown in Figures A.2 and A.3. The goal of including these questions is to try to ascertain a link between past experience and the ultimate choice of project, to determine if TEASync is achieving its goal of allowing inexperienced developers to create multi-user applications and games.

Based on the user's input about which types of games they had experience developing, Section 3's questions were automatically tailored to ask either their experience or their perception of the experience (for instance, the actual vs. perceived difficulty). Figure 6.2 shows an example of this question for those who said they had made single player games and had not made multiplayer games before. The full questions with the four cases of single player/multiplayer experience are shown in Figures A.4 (Y/N), A.5 (N/Y), A.6 (Y/Y), and A.7 (N/N).

Finally, the Section 4 of the pre-survey contains a Likert scale question to collect students' baseline opinions of computer science and the course (Figure 6.3). The full section 4 is shown in Figure A.8.

11. How many computer science courses did you take in high school? 


☐ 0

☐ 1

☐ 2

☐ 3

☐ 4 or more

12. How often do you work on your own programming projects (outside of school)? 

☐ Every day

☐ 2-3 times per week

☐ Once per week

☐ 2-3 times per month

☐ Once per month or less

☐ Never

Figure 6.1: Examples of computer science experience questions in the pre-survey.  
The full pre-survey Section 2 is shown in Figures A.2 and A.3.

17. You indicated that you **have** created a **single player** game before. For each of the following aspects, rate the difficulty you experienced while creating your **single player** game. ☐☐

	Very Difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Figure 6.2: Example past game development question from Section 3 of the pre-survey. This example asks students who had made a single player game before about their experienced difficulties doing so. The full questions are in Figures A.4, A.5, A.6, and A.7

19. For each of the following statements, please read the statement carefully and rate your level of agreement with the statement. ☐☐

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
I don't feel that I understand what Design Thinking is	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
It is important for software professionals to understand user needs	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Computer science and coding are forces that can be used for evil	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I prefer to code when it has a bigger purpose	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
It is difficult to know what my user's needs are	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I enjoy programming	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I enjoy programming in Elm	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I am excited about the computer science program	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I wouldn't like to pursue a career in computing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Figure 6.3: Pre-survey Likert scale question to ascertain students' baseline experience with computer science. The full pre-survey Section 4 is shown in Figure A.8.



### 6.3.2 Post-Implementation Survey

The Post-Implementation survey was focussed on understanding the students' choice of whether to create a single or multiplayer game, their reasons for doing so, as well as their general opinion of tools like the TEASync framework, STaBL.Rocks, and the state diagram tool.

The Post-Implementation Survey's questions are shown in Appendix A.2. Figure A.9 again shows informed consent questions, including a question allowing respondents to answer Pre-Survey questions if they did not previously answer the survey, and still receive credit for it in terms of the gift card draw.

Section 2 was designed to ascertain students' reasons for choosing to make a single player or multiplayer game (see Figure 6.4), and their experienced difficulties during the course, with the same options as Section 3 of the pre-survey. Similarly to the pre-survey, the questions were tailored based on whether they answered that they had made a single player or multiplayer game. The full Section 2 of the post-survey is available in Figures A.10, A.11, and A.12.

Finally, Section 3 asked the students some Likert scale questions about their experience in the project and course, and their overall impression of the project (see Figure 6.5). The full Section 3 is shown in Figure A.13.


## 6.4 Focus Group Design

The focus group was designed to get more in-depth information from the students about their experience in the course and using the TEASync framework. Due to ethical considerations, focus groups were not run by the instructor or TAs of the course


10. You indicated that you chose to make a **single player** game for your project. Thinking about your project this semester, please rate how much each of the following statements applies to you and your group. ☐

	Does not apply to us	Somewhat applies to us	Applies to us	Completely applies to us
We enjoy playing single player games, so we wanted to make one	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought it would be more interesting to create a single player game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought the resulting game would be more engaging being single player	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought we could create a better project by making a single player game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought we could get a better mark by creating this kind of game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We had more experience making single player games	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We wanted to try something new	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought a single player game would be more applicable to the Design Thinking problem	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Figure 6.4: Post-survey Section 2 question asking reasons why the student decided to choose single player or multiplayer for their project. The full post-survey Section 2 is shown in Figures A.10, A.11, and A.12.

12. For each of the following statements, rate your agreement with the statement. 

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
I feel everyone in my team contributed equally to the project	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The State Diagram is a useful tool for creating single player games	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The State Diagram tool was difficult to learn	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The State Diagram tool was powerful enough to do what we wanted to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The TEASync framework is a useful tool for creating multiplayer games	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The TEASync framework was difficult to learn	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The TEASync framework was powerful enough to do what we wanted to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The <a href="#">STaBL Rocks</a> platform made it easy for us to collaborate	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The Design Thinking components of the project were more difficult than the technical (coding) aspects of the project	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Our group is happy with our decision to make the type of game we made (single player or multiplayer)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

13. Compared to other course projects, how much did you enjoy the 1XD3 project? 

	-2 - Did not enjoy at all	-1 - Did not enjoy	0 - Indifferent	1 - Enjoyed	2 - Enjoyed very much
Project enjoyment	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Figure 6.5: Post-survey Section 3 Likert scale-based question about course and tool experiences and overall project experience question. The full post-survey is shown in Figure A.13.

(including Dr. Anand and Christopher Schankula), but were run by collaborators. The identities of those who participated were not recorded, and the transcript was de-identified before being given to the author of the current work or Dr. Anand.

Here are some selected questions from the focus group:

- What parts of the project did you enjoy? Why?
- For those who chose to make a [single player/multiplayer] game, what reasons did your group have for choosing to do so?
- Which aspects of the project were difficult for you?

The full list of focus group questions is included in Appendix A.3. Questions were centred around the students' experiences in the course and using the tools, with the goal of enriching the data from the surveys. Some questions centred around the DT process were out of the scope of the current work.

## 6.5 Compilation Statistics Design

Students were given the option to opt into the use of STaBL.Rocks compilation statistics for the Fall 2023 and Winter 2024 school term. These statistics include the following:

- Mean and median number of modules created per student
- Mean and median number of times each student compiled their code
- Mean and median number of lines of code per compile
- Mean time-fairness of compiles per student (see Section 6.5.1)

- What types of multiplayer features students implement using TEASync (full multiplayer game, single player with shared features like leaderboards, or no multiplayer features)

The goal of these statistics is to ascertain if there was a measurable difference in the coding habits of those who chose to create multiplayer game versus single player. Due to limitations of ethics, unless all students in a given group opted into use of data, it was not possible to analyze the code of an entire group. Thus, the analysis will be focused on students, despite it being a group-based project.

While single productivity metrics like lines of code or number of compiles on their own have been shown to be problematic [33], we hope they will provide a general comparison between the single player and multiplayer group.

### 6.5.1 Time Fairness Metric

One metric of engagement is how consistently students were engaging with the Integrated Development Environment. We have developed a *fairness metric* [56] that can be applied to a list of data to compute a score from 0 to 1, according to how spread out that data is. It was originally developed to compare fairness amongst teammates' work habits, but substituting teammates for time gives us a time-fairness metric. To compute this, we start with an *unfairness metric*:

$$\text{unfairness}(C) = \frac{\sum_{c,x \in C, c > x} (c - x)}{(|C| - 1) \cdot \sum_{c \in C} c}$$

where  $C$  is a multi-set representing the number of compiles<sup>1</sup> each day from September

---

<sup>1</sup>The fairness metric can take in any set of data as input, but we have chosen the number of compiles per day to compute a measure of engagement

5th, 2023 to December 20th, 2023 (the date of the final exam) and from January 8th, 2024 to April 18th, 2024.

The metric is 0 (meaning completely unfair) if a student compiled an equal number of times per day, and 1 if all the work was done on one day ( $n$  compiles done on one day, and 0 on the other days). Values between 0 and 1 correspond to the proportion of work done on a day when doing it on another day would be more fair. For instance, if the student did work on three days and did 10, 5 and 5 compiles each day, the unfairness metric would be 0.25, since they did 5 extra compiles on day 1 compared to the other two days.

Then, we compute the fairness metric as  $\text{fairness}(C) = 1 - \text{unfairness}(C)$ . In our example, the fairness value would be 0.75.

## 6.6 Threats to Validity

There are several potential threats to validity worth noting, including, but not limited to:

- With regards to the surveys, focus group, and the compilation statistics there is a self-selection bias, which may skew the results since students who decide to participate may not form a representative sample.
- Focus groups responses may be skewed by peer pressure since they are speaking in front of a group of their peers. The focus group format itself may preclude some students from taking part in the first place.
- Since compilation statistics are only accessible per-person due to ethics and informed consent limitations, it may be difficult to draw conclusions about an

entire group based on individual opt-ins.

- Compilation statistics may be flawed if students decided to do part of their development on a different IDE where statistics are not able to be collected.
- The sample size for the surveys and focus groups may preclude having enough information to draw certain fine-grained conclusions.
- Due to ethics limitations, there is no traceability between responses in the surveys and what was said in the focus groups.
- Pre-survey questions were asked again during the post-survey in case someone wished to respond despite not having done so before. These responses may not be comparable to those who answered it earlier in the course, since their experiences would be different than the students who did it near the beginning.
- The word “Very” was mistakenly replaced with the word “Extremely” in the post-survey difficulty questions, which may have skewed the difficulty results in the course versus the previously experienced and perceived difficulty ratings.
- Difficulty ratings, as with many other questions in the survey, may be perceived by different respondents differently.
- Metrics like average lines of code, total compiles, and the number of modules may not be a good measure of engagement as more compiles for instance may be an indicator that students were struggling and had to do more trial-and-error.
- Metrics like average lines of code, total compiles, are an indication of engagement but there is no way to know if the choice of game type caused the increase in engagement or not.

- The time-fairness metric gives a relative idea of how spread out the work was, but compiles may not be a perfect measurement of work done. It is also normalized by the total amount of work done: someone may do more work than someone else but have a lower time-fairness metric. We could not apply this metric to teams since teams may not have all opted into data use.
- Some students may have used the multiplayer framework to make a game with no real multiplayer features, or some may have used it to simply create a shared feature like a leaderboard. Depending on their definition of single vs. multiplayer game, they may have answered differently.



# Chapter 7

## Results

This chapter contains results of the usability study described in Chapter 6 as well as some example TEASync applications made by students in their project.

Section 7.1 describes results from the pre- and post-implementation surveys. This is followed by focus group results in Section 7.2. Section 7.3 contains results of the data from the STaBL.Rocks code compilation analysis. Finally, analysis and screenshots of some example TEASync applications developed by students and used with permission are shown in Section 7.4.

### 7.1 Survey Results

In total, there were 26 responses to the pre-survey and 22 to the post-survey, out of a class of 198 students. Of the 22 who answered the post-survey, 12 answered that they were new respondents and wished to answer the pre-survey questions as part of the post-survey. Thus, only 10 were returning from the original pre-survey. This means a total of 38 unique individuals answered our surveys, for a response rate of 19%. In

total, 18 students attended the two sessions of identical focus groups, a participation rate of 9%.

### **7.1.1 Pre-Implementation Survey**

This subsection details the results of the pre-implementation survey, for which questions are shown in Figure A.1.

#### **Demographics Results**

Figure 7.1 shows the demographic results from the pre-survey as well as the students who did the pre-survey questions as part of the post-survey. In total, there were 38 students. Being a first-year computer science course, most of the students fell into that year of study and program. The approximately 1/4 of female-identifying students is higher than the demographics of the program as a whole. A wide array of programming experience was recorded.

#### **Perceived vs. Experienced Game Creation Difficulty**

Figure 7.2a shows the results of the question “Have you ever programmed your own game?”. Based on their answer to this question, they were asked for their experienced or perceived difficulty with several aspects of creating a game.

To justify the need for a framework to improve the accessibility of multiplayer games, students were asked what their prior experienced or perceived difficulty with each type of game was. Students who stated they had previously made that type of game before were asked about their experienced difficulty, whereas students who had not were asked about their perception of difficulty. Difficulty was rated on a scale of very difficult, difficult, neither difficult nor easy, easy, and very easy, which were

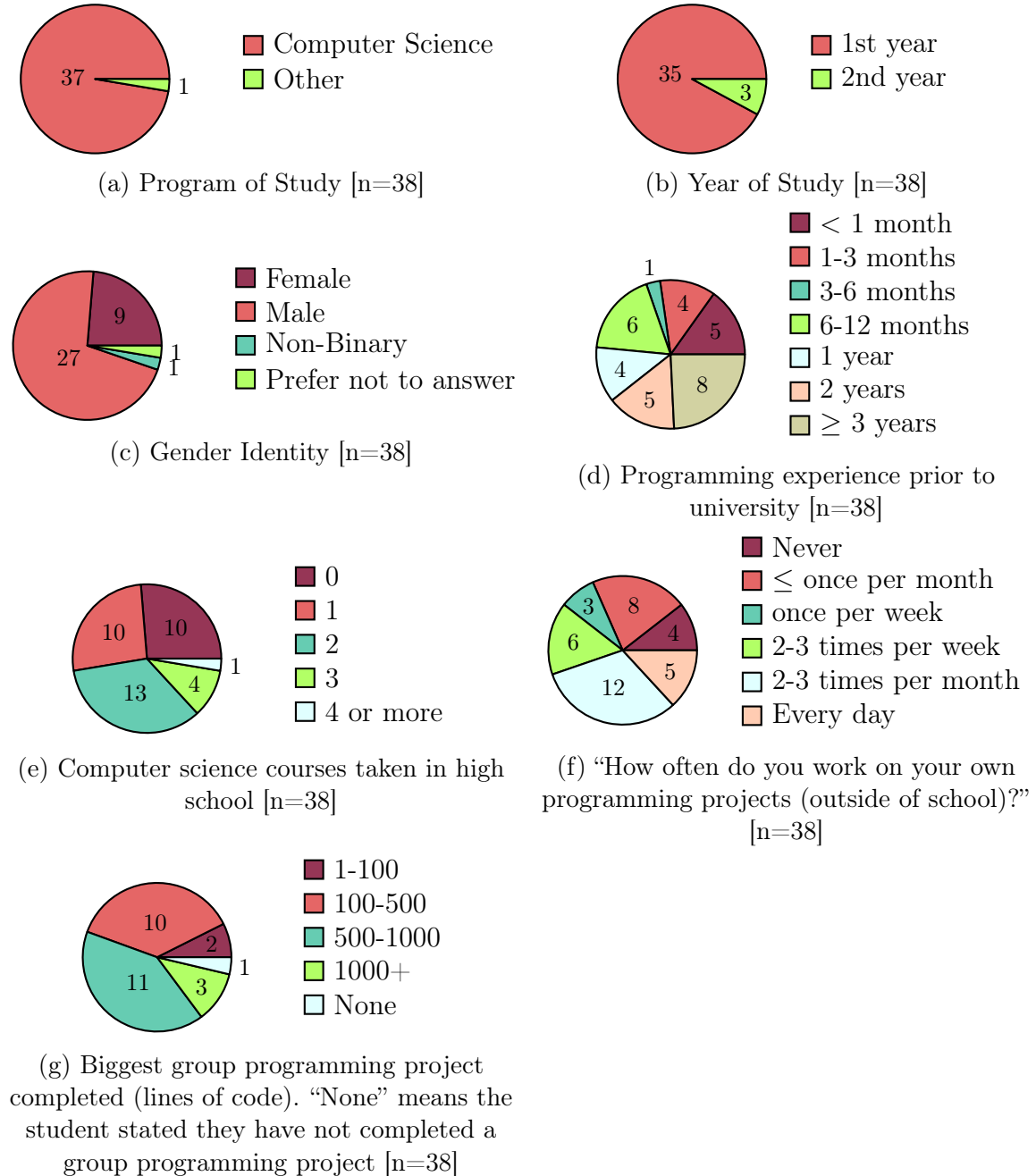
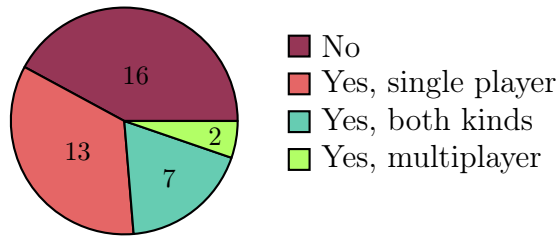


Figure 7.1: Demographic results. All results are in counter-clockwise order of the legend, with the first one starting on the positive x-axis.



(a) “Have you ever programmed your own game?” [n=38]

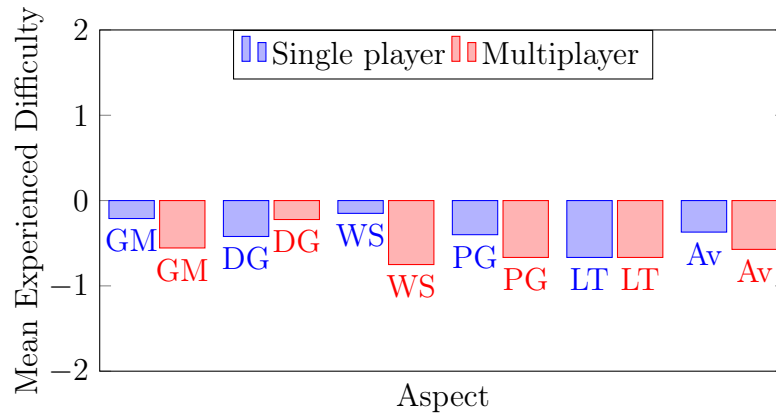
Figure 7.2: Results for questions about prior experience programming single player and multiplayer games.

assigned integer values of -2 to 2. They were asked to rate the experienced or perceived difficulty of five aspects: designing game mechanics, designing graphics, writing the storyline/script, programming/coding the game, and learning the tools needed to make the game. Figure 7.3 shows the results for single player and Figure 7.3b shows the results for multiplayer.

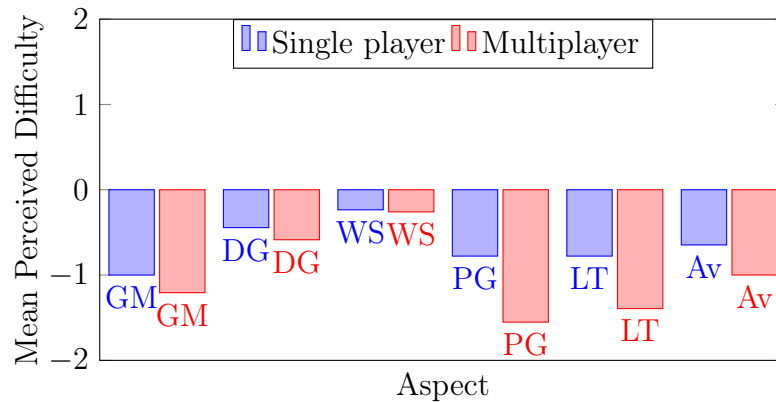
The rated difficulty of most aspects was higher for multiplayer games, and the perceived difficulty was almost universally higher than the experienced difficulty. Of note is that the programming and learning tools were perceived as much higher for multiplayer, which suggests a need for better tools to help new programmers feel confident making multiplayer games.

### Likert Scale Results

The results of the pre-survey Likert Scale questions are shown in Figure 7.4. Questions were purposely mixed between “positive” and “negative” wording to avoid respondents selecting the same answer for all questions in sequence. Of note to the Design Thinking nature of the project is that students responded in agreement to questions like “It is important for software professionals to understand user needs” and “I prefer to code when it has a bigger purpose”.



(a) Reported prior experienced difficulty



(b) Reported perceived difficulty

Figure 7.3: Reported prior experienced and perceived difficulty of aspects of single player vs multiplayer games. -2 = “Very difficult”, -1 = “Difficult”, 0 = “Neither difficult nor easy”, 1 = “Easy”, 2 = “Very easy”. GM = “Designing game mechanics”,

DG = “Designing graphics”, WS = “Writing the storyline/script”, PG = “Programming/coding the game”, LT = “Learning the tools needed to make the game”, Av = “Mean”

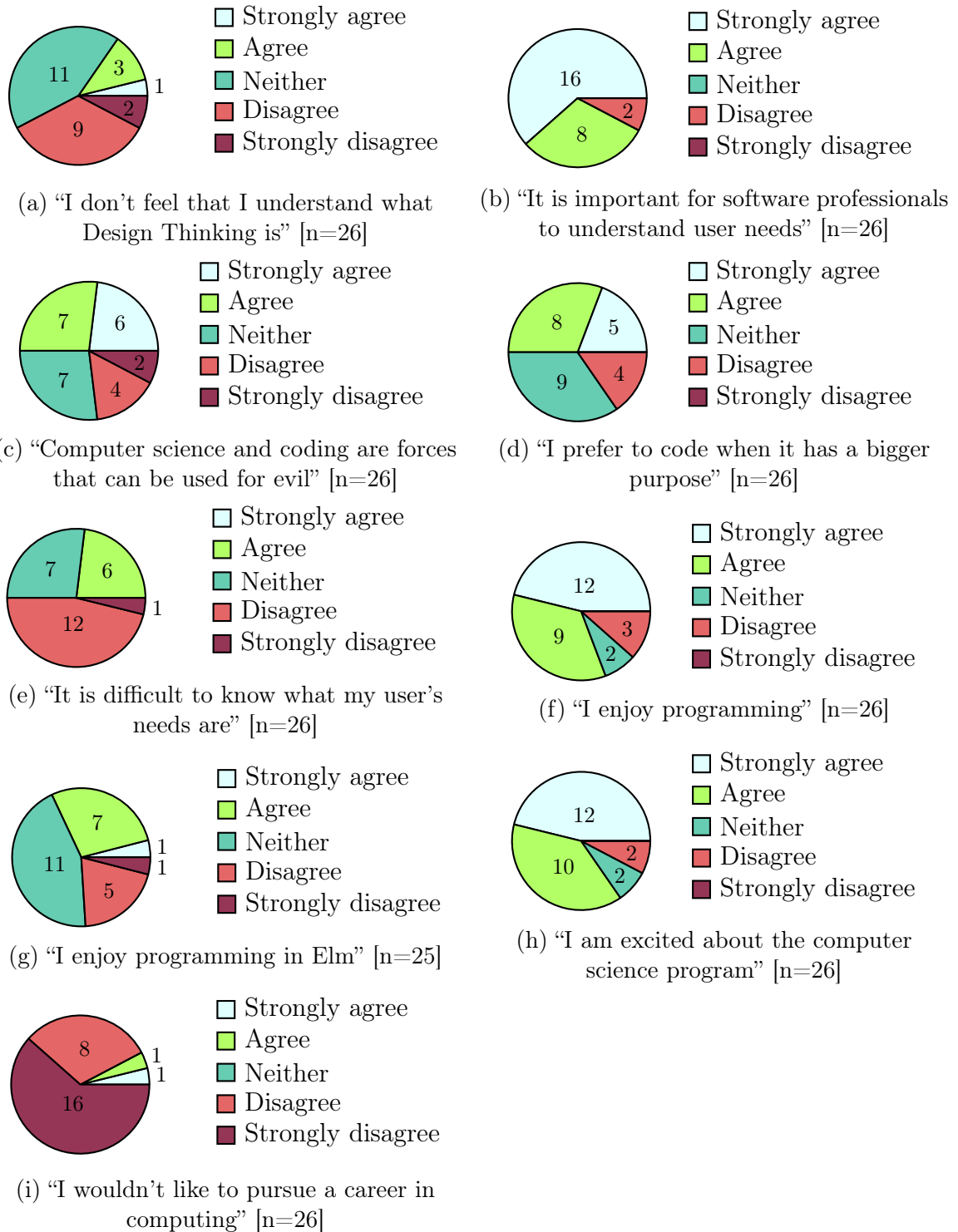


Figure 7.4: Likert Scale (pre-survey) results. "Neither" is shorthand for "Neither agree nor disagree"

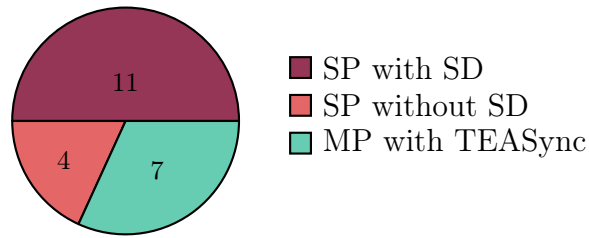


Figure 7.5: “For your project, did you choose to create a single-player game or a multiplayer game?” [n=22]. SP/MP = “Single player”/“Multiplayer”, “SD” = “State Diagram Tool”

### 7.1.2 Post-Implementation Survey

This section shows the results for the post-implementation survey. In total, there were 22 responses to the post-implementation survey. Of those, 12 were new respondents who didn’t previously complete the pre-survey but wished to do so at that point, and 10 were returning respondents.

#### Type of Game Chosen

Figure 7.5 shows the type of game respondents said they made with their team. This question (shown in Figure A.10) asked if they made a single player game using the State Diagram (SD) tool, a single player game without the state diagram tool, or a multiplayer game using TEASync. Approximately 2/3rds of respondents made a single player game, with almost 3:1 opting to use the SD tool.

#### Reasons for Choosing Single player vs. Multiplayer

Figure 7.6 shows respondents’ reported reasons for choosing to make a single player vs. multiplayer game. Table 7.1 shows the acronyms used as shortforms in this figure. These correspond to the statements shown in Figure A.11 and A.12.

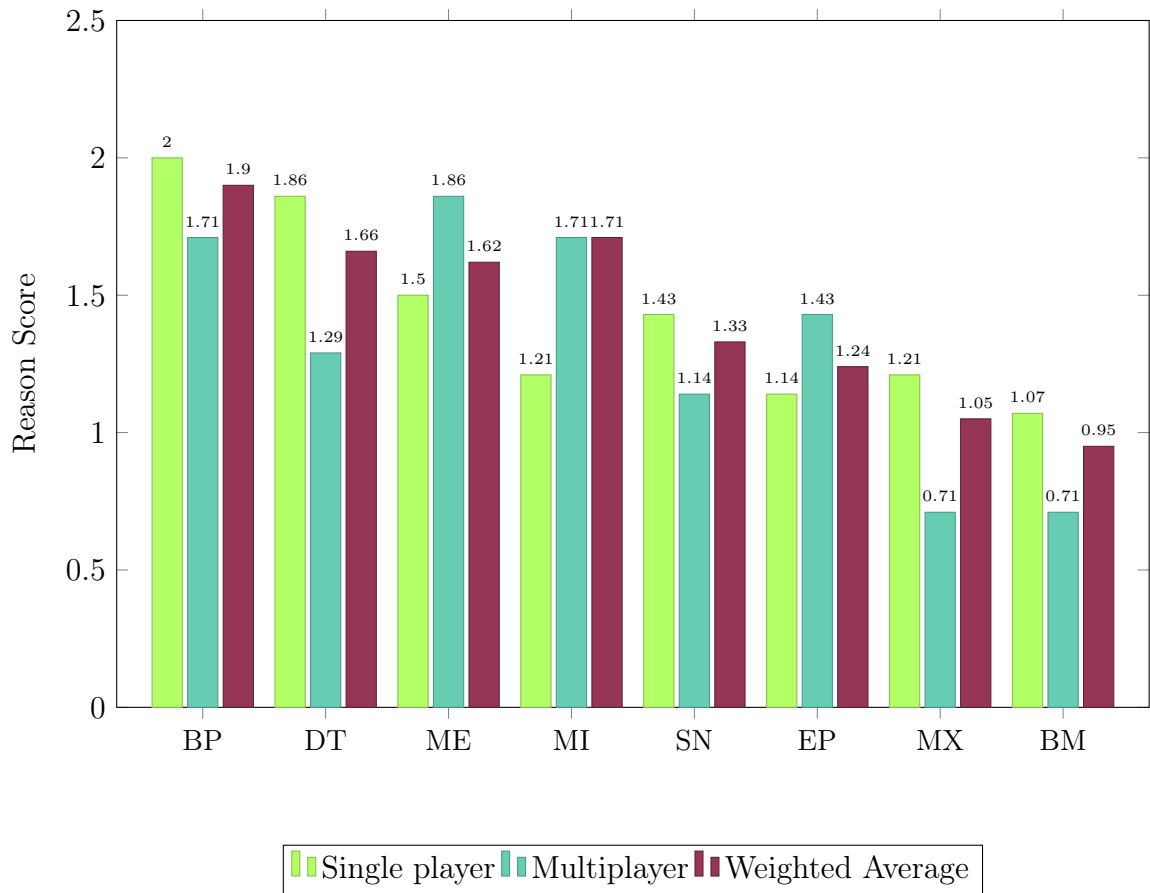


Figure 7.6: Reported reasons for choosing single vs. multiplayer games [n=21]. Table 7.1 expands the acronyms used here. Reported here are the mean values from a Likert scale question of agreement with the statement.



Table 7.1: Shortforms of factors affecting single player/multiplayer choices

Shortform	Likert Scale Statement
BP	We thought we could create a <b>Better Project</b> by making a [single player/multiplayer] game
DT	We thought a [single player/multiplayer] game would be more applicable to the <b>Design Thinking</b> problem
ME	We thought the resulting game would be <b>More Engaging</b> being [single player/multiplayer]
MI	We thought it would be <b>More Interesting</b> to create a [single player/multiplayer] game
SN	We wanted to try <b>Something New</b>
EP	We <b>Enjoy Playing</b> [single player/multiplayer] games, so we wanted to make one
MX	We had <b>More eXperience</b> making single player games
BM	We thought we could get a <b>Better Mark</b> by creating this kind of game

### Reported Difficulties of Making Games During the Course

Figure 7.7 shows the reported difficulty level for aspects of game making, shown divided between students who reported making single player and multiplayer games. As before, this was rated on a scale from -2 to 2, which corresponded to “Extremely difficult” to “Extremely easy”.

### Distribution of Work Amongst Teams

Figure 7.8 shows the results to the Likert scale rating of the statement “I feel everyone in my team contributed equally to the project”, split into students who did single player vs. multiplayer. Single player teams were more likely to report that work was more evenly distributed, though in both cases the results suggest some unfairness.

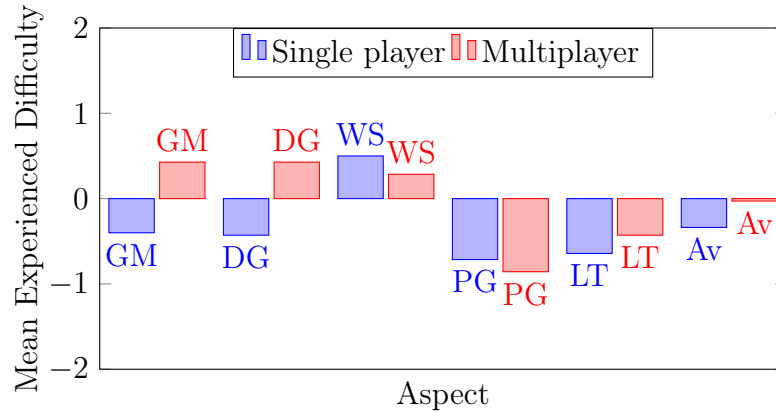


Figure 7.7: Reported experienced difficulty of aspects of single player vs multiplayer games in the course. -2 = “Extremely difficult”, -1 = “Difficult”, 0 = “Neither difficult nor easy”, 1 = “Easy”, 2 = “Extremely easy”. GM = “Designing game mechanics”, DG = “Designing graphics”, WS = “Writing the storyline/script”, PG = “Programming/coding the game”, LT = “Learning the tools needed to make the game”, Av = “Mean”

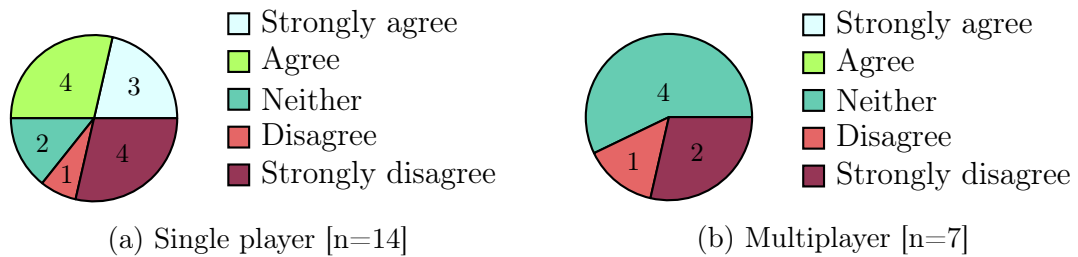


Figure 7.8: Responses to statement “I feel everyone in my team contributed equally to the project” for single player vs. multiplayer game creators.

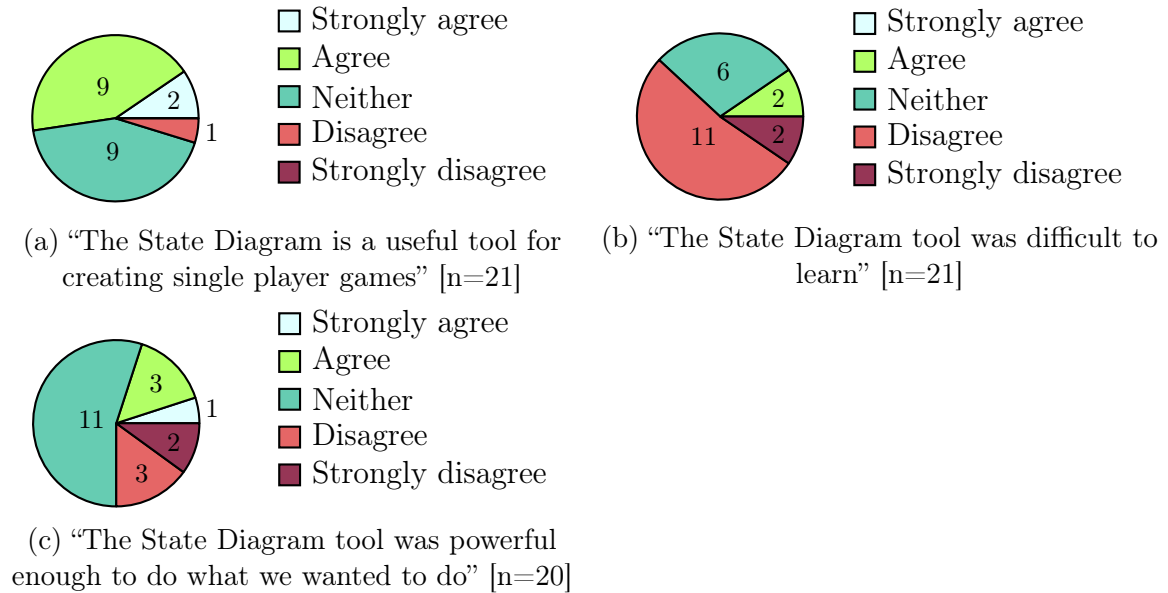


Figure 7.9: Responses to Likert scale statements about the State Diagram tool.

### Usefulness, Difficulty, and Power of State Diagram vs. TEASync

Figure 7.9 and 7.10 show Likert scale answers to statements about the State Diagram tool and TEASync framework. The data is approximately equal in terms of usefulness, but students were more likely to report that TEASync was more difficult to learn, but also that it was powerful enough to achieve their goals.

### Happiness with Decision to Create Single player vs. Multiplayer

Figure 7.11 shows the reported agreement with the statement "Our group is happy with our decision to make the type of game we made". TEASync teams were slightly more likely to report being happy with their decision in the end.

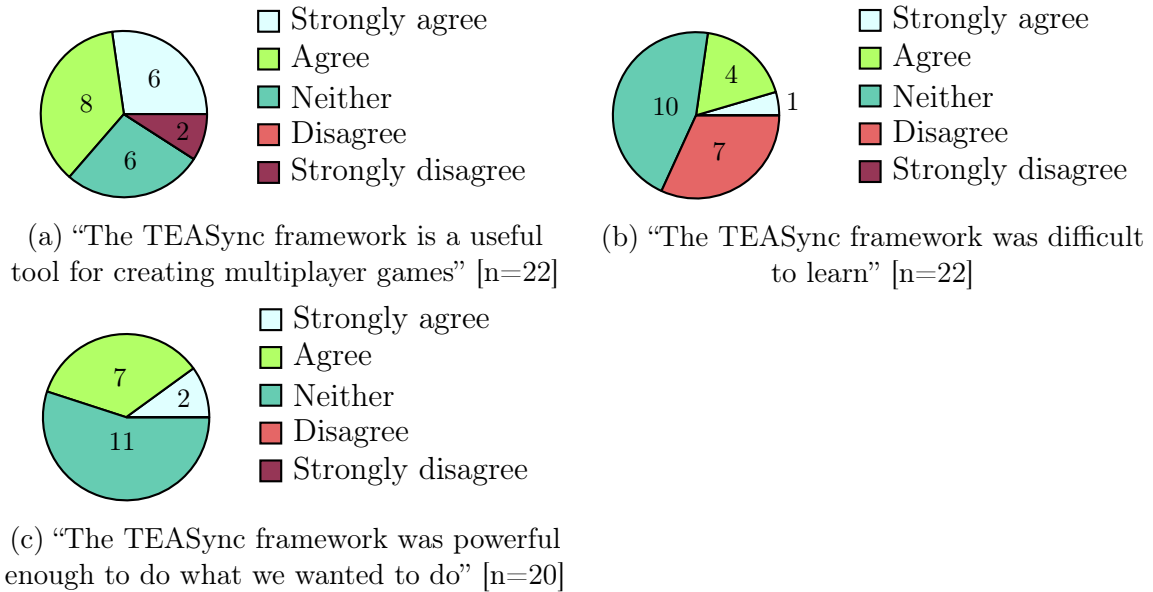


Figure 7.10: Responses to Likert scale statements about the TEASync framework.

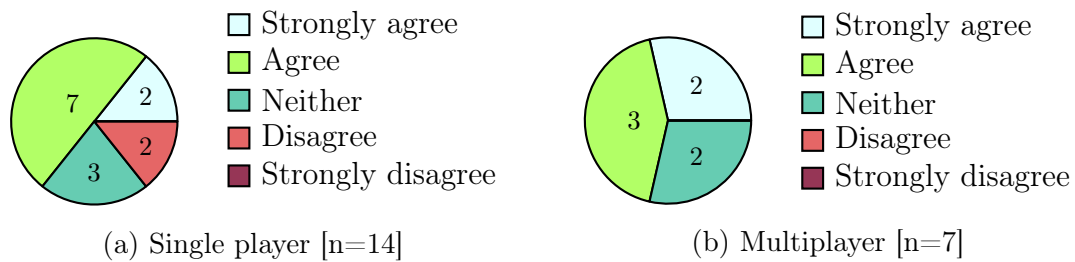


Figure 7.11: Responses to statement “Our group is happy with our decision to make the type of game we made”, for single player vs. multiplayer game creators.

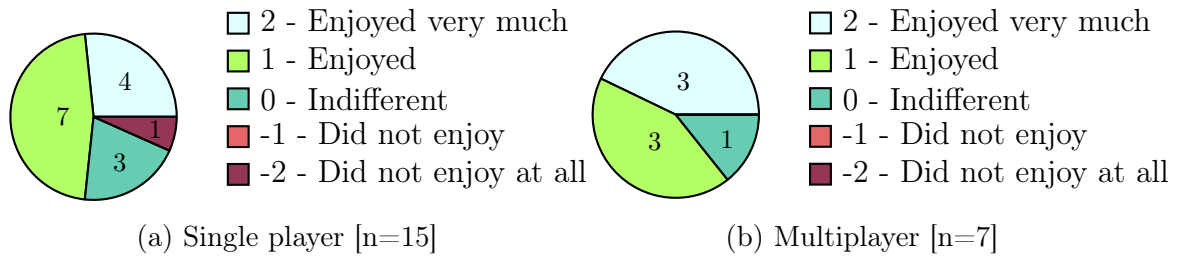


Figure 7.12: Responses to “Compared to other projects, how did you enjoy the 1XD3 project?” for single player vs. multiplayer game creators.

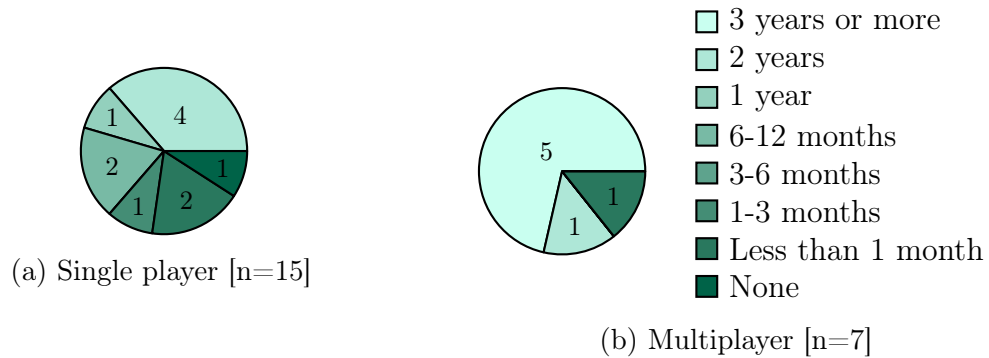


Figure 7.13: Programming experience versus choice of single/multiplayer

## Overall Enjoyment of Project

Figure 7.12 summarizes the responses to the question “Compared to other projects, how much did you enjoy the 1XD3 project?”. Students who chose to go multiplayer were more likely to report a higher overall enjoyment of the project.

## 7.2 Focus Group Results

In total, 18 students participated in the focus groups, for a participation rate of 9%. This section provides a summary of findings that help to give some context and depth to some of the survey questions. Quotes have been edited to remove filler words such as “like” and “umm” where it does not change the meaning.

### 7.2.1 Barriers to Choosing Multiplayer Games

Students were asked their reasons for choosing single player versus multiplayer games for their project.

TEASync was introduced into the course about 4 weeks in, and this seems to have prevented some students from choosing to use it. One big sentiment echoed was that the timing of introducing TEASync affected their decision: “the main reason that we wanted to do a single player game is because we had already designed the game before we learned about the capabilities of TEASync.” Another student echoed this sentiment: “I would like to do multiplayer [...] if it was introduced a bit earlier on”.

Another barrier was the perceived difficulty of the model, which hampered some students, especially if they were newer to coding. One student stated they thought it “favours students who have had a bigger background in computer science” which made them feel they “didn’t even know what to tip my toes in it because of the the complex nature it involved.” Another student said they were going to try to “make our game to be multiplayer after the full system was introduced to us, but then we found out it’s actually way more complex and will increase our workload.” This shows that the LG-MVU model used for concurrency is still perceived as too difficult, stopping some students from choosing to use the framework.

The logistics of handling debugging a multiplayer application was also stated as a barrier: “Those things are more complex [...] the server we need to compile it several times. We are not the one who are maintaining the server, but at that process of compiling and going back through your debugging and everything.” While the student acknowledged that the servers can be deployed automatically, this process was still found to be too complicated.

A lack of documentation was cited as difficulty for the groups as well: “Would have been really useful to have more documentation in some sense on the things that are doable or possible with TEASync. And so I think the lack of documentation in some sense restricts us from knowing what’s there and what’s kind of available to us”. This sentiment was echoed for Elm itself: “when we are not with our instructor like just very hard to find resources online to figure out how this stuff ourselves.”

Finally, students also shared that they chose to do single player based on their DT interviews: “One of the reasons for that is we found in our interviews that multiplayer aspects aren’t always very popular among, some people obviously enjoy them, but for other people they prefer to find social interaction through their games as an optional thing.”

### **7.2.2 Reasons for Choosing Multiplayer**

On the other hand, some groups decided to pursue a multiplayer project based on interviewee feedback: “We found that the interviewees really liked the social aspect, and we thought it would be a good idea for older people, especially seeing [...] what happened during COVID [...]”.

One student praised TEASync’s ability to spin up servers on the fly for testing: “I thought it was super cool that they gave us servers, so I didn’t need to set anything up. They just click and run. It was buggy at first, but then I just click the button and it just works and then I could give them a link that. It’s super cool hey now it’s on your computer and now we play. That’s kind of cool to see something you build come to life and see people actually enjoy the products you make.”

A student stated that they enjoyed testing their multiplayer games, given their

interactive nature: “I quite enjoyed playing the games that we made [...] I ended up testing with my family.” They jokingly added, “They gave me a lot of negative feedback. My mom is my biggest hater, but yeah, it was really fun nonetheless.”

While some students found TEASync to be complex, others were pleasantly surprised by its simplicity: “Yeah, I thought at the start learning TEASync would be very difficult, like, it’d be very difficult to kind of get started with it. But I was pleasantly surprised at how easy it was to actually get started with it,” adding that “It just built off of what we were using before with the single player into using it for multiplayer” and that “I enjoyed using the multiplayer in the global model in this project.” One student found the LG-MVU model to be a good introduction to multi-user applications: “I thought it was actually a pretty good introduction to the idea of a local versus global models in multiplayer games and just like was a good introductory understanding of how they worked.”

Finally, a student remarked that creating a multiplayer game was more satisfying than a single-player game: “One thing I would note is that this concept of satisfaction for having things work properly in the game. It definitely feels as though it is more present in the multiplayer framework because generally in multiplayer games they are a lot more concrete because you generally play them with other people, which means that since it’s shared with others, it’s more relevant than single player applications would be.”

## 7.3 Code Compilation Statistics Results

A total of 36 students opted into use of data. Of those 36 students, 20 also filled out the the post-implementation survey and therefore specified which type of game their



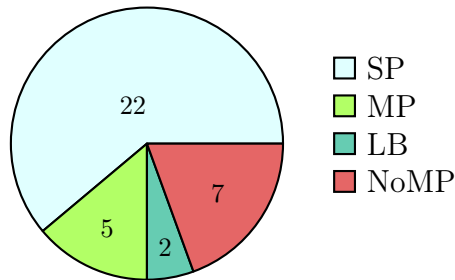


Figure 7.14: Single player vs. use of TEASync framework [n=36] (SP=Single player game, MP = TEASync game with multiplayer features, LB = Single player game with TEASync leaderboard, NoMP = TEASync game without multiplayer features).

group made. The other 16 did not fill out the post-implementation survey so their group’s choice was gleaned from the STaBL.Rocks data. The data was used to look more specifically into uses of the multiplayer framework. For instance, some teams used TEASync but did not include any multiplayer features in their application. Anecdotally, students reported that they wanted to use TEASync so they would have the option to add multiplayer features later. Others used TEASync to make a single player game that had shared data like a leaderboard, which was an example given in class.

Table 7.2 summarizes the compilation data for both types of students: students who made a single player game (SP) and those who made a TEASync game (MP). Statistics are split across semester 1 (S1) and semester 2 (S2). Most metrics are higher for students who decided to make a multiplayer game for their project. One notable exception is the median lines of code per compile, which is smaller despite the fact that the mean is higher. The mean and median time fairness values were remarkably consistent at around 0.13, indicating a high concentration of work, likely around due dates and the twice-weekly lab days. There is a slight uptick in the mean time fairness for the MP group in semester 2, indicating that some enthusiastic

students were highly engaged by the multiplayer framework, which is consistent with other data that multiplayer students skewed towards higher levels of programming experience.

## 7.4 Student TEASync Applications

This section includes some examples of TEASync applications developed by the students in the class, as well as some statistics about them. Table 7.3 summarizes some statistics for these three applications: *Garlic Phone*, *Tap Scotch*, and *Brushstroke Journey*.

### 7.4.1 “Garlic Phone” Game

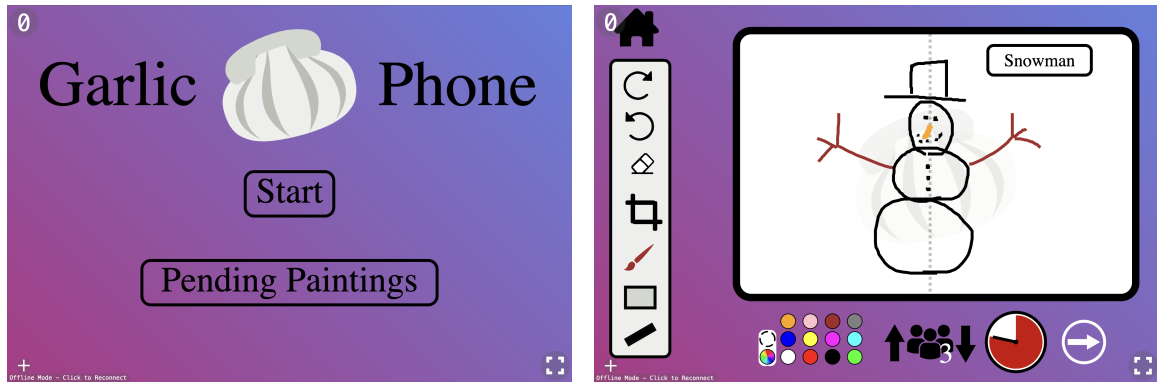
The Garlic Phone game (Figure 7.15) is a multiplayer game where players draw images, have them divided in half, and then other players take turns completing the drawing. The game has a complex local portion in comparison to its global part, which was for sharing the drawings amongst players. In this way, this example used TEASync to facilitate a shared experience. Sharing the vector graphics requires many floating-point numbers and this app’s network efficiency therefore benefits greatly from the binary encoding format.

### 7.4.2 “Tap Scotch” Game

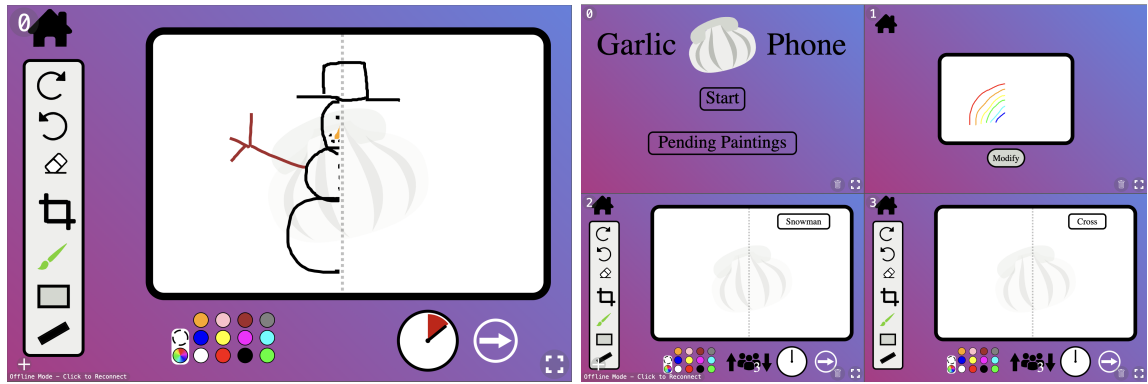
Tap Scotch is a multiplayer game created by another team of first-year students. Players compete to tap the randomly-generated tiles as fast as possible. Players can see where their opponent is as an outlined tile if they are behind. The game is

Table 7.2: Data from STaBL.Rocks IDE [n = 36 users]

Statistic	Semester	SP	MP	All
Mean Compiles	S1	3511	3494	3505
	S2	3362	3645	3472
	Both	6873	7140	6977
Median Compiles	S1	2739	2729	2739
	S2	2696.5	2866	2785
	Both	5709	6097.5	5782
Mean Modules Created	S1	51	53	52
	S2	134	179	152
	Both	185	233	204
Median Modules Created	S1	52.5	49.5	51.5
	S2	120.5	153	127.5
	Both	171.5	217	179.5
Mean LOC/Compile	S1	163	146	157
	S2	278	360	310
	Both	233	252	240
Median LOC/Compile	S1	140.5	128	134.5
	S2	218	219.5	219
	Both	166.5	155.5	162.5
Mean Time Fairness	S1	0.126	0.129	0.127
	S2	0.133	0.139	0.135
	Both	0.130	0.130	0.130
Median Time Fairness	S1	0.126	0.130	0.127
	S2	0.131	0.129	0.130
	Both	0.129	0.130	0.129



(a) The main menu allows players to play or see the other players' drawings. (b) Users can draw an image, which will be sent to all the other players.



(c) Once the image is split in half, other players try to complete the image. (d) Multiple players can play together, sharing the images amongst themselves.

Figure 7.15: Screens in the *Garlic Phone* application.

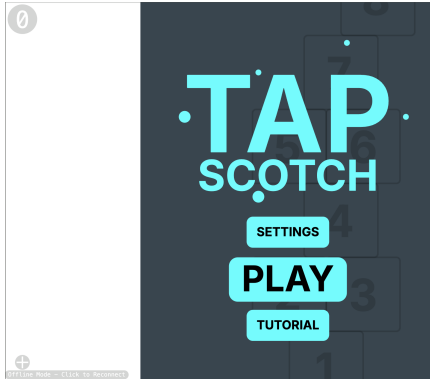
Table 7.3: Selected App Statistics

Statistic	GarP	TapS	BruJ	Counting Example
Main Module LOC	1233	847	907	46
Types Module LOC	261	45	65	30
Total LOC	3848	892	972	76
Custom Types	27	5	2	1
Custom Modules	20	0	0	0
LocalMsg Constructors	30	10	28	1
LocalModel Fields	40	10	27	1
GlobalMsg Constructors	3	8	1	1
GlobalModel Fields	2	5	9	1

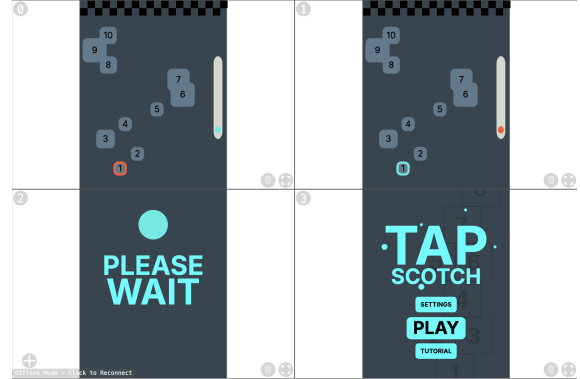
intended to measure reaction times of players. The game includes a lobby system that allows multiple pairs of opponents to play on the same server instance at the same time. Figure 7.16 shows some screens from the game.

### **7.4.3 “Brushstroke Journey” Game**

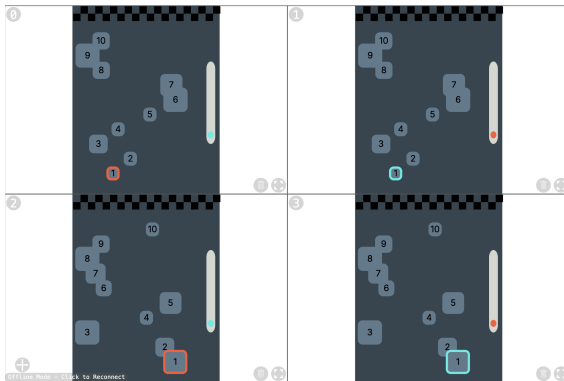
Brushstroke Journey is another game created in the class, where players travel through the game world and complete drawings to advance to the next part of the game. The game is intended to measure the accuracy of the players’ mouse movements. This game, while largely single player, utilizes TEASync to share leaderboard data. Figure 7.17 shows some example screens from the game.



(a) The main menu allows players to access a tutorial, settings and the game itself.



(b) The game has a lobby system which allows any number of pairs of clients to create games together. In this case, player 3 is waiting for player 4 to join, while player 1 and 2 play concurrently.

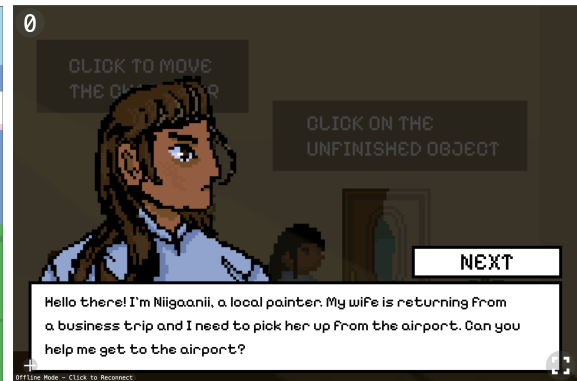


(c) Once the second pair joins, they can play their own instance of the game concurrently to the first pair.

Figure 7.16: Screens in the *Tap Scotch* application.



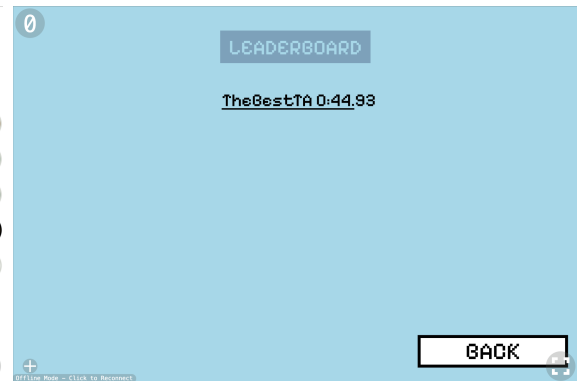
(a) The main menu allows players to play the story mode, creative mode, or log out.



(b) The story mode takes players through a story that is completed by filling out incomplete drawings.



(c) Players paint the incomplete drawings according to hints given in gray.



(d) Players can submit their scores to a leaderboard using TEASync.

Figure 7.17: Screens in the *Brushstroke Journey* application.

# Chapter 8

## Conclusions

This chapter presents some conclusions from the work, ways in which the data can be interpreted to answer the research questions, and the next steps and future research directions.

### 8.1 Summary

TEASync successfully implements a functional, event-driven framework for creating multi-user applications and games. The framework leverages Elm as the client language, which provided useful properties like a pureness guarantee, strong typing, and its model-view-update architecture. Combined with Software Transactional Memory (STM) in the backend Haskell language, this helped to ensure consistency amongst the clients. We demonstrated that this framework can be used by first-year computer science students to build non-trivial applications with multi-user functionality.



## 8.2 Research Questions

This section answers the research questions discussed in Section 1.4. Some of these questions were answered earlier in the thesis, while some can be answered at least in part from the data presented in Chapter 7.

### 8.2.1 RQ1. How can EDP in a functional context be extended to support multi-user applications?

This work shows that the model-view-update paradigm can be extended to work for multi-client applications, provided that the system can guarantee the ordering of messages delivered to clients. Several different ways of doing so were compared in Chapter 4. MVU can be implemented in a shared fashion directly, allowing all the messages (and therefore the entire model) in an MVU application to be shared amongst all clients. However, a useful abstraction is to separate out local and global portions to allow some state to be local to each client, which led to the creation of LG-MVU.

Section 4.4 discusses several different schemes for maintaining consistency, though some are more likely to lead to race conditions manifesting as lost message updates. Message-based synchronization with fast-forwarding was chosen as the scheme to be implemented due to its low complexity, low network traffic requirements, and good concurrent properties.

Elm’s pureness guarantees, strong typing and total functions were helpful in ensuring several properties. Section 5.6 discusses how this pureness helps to ensure that

no bugs “leak in” from outside due to side-effects, which is helpful for new programmers and for testability in general. Furthermore, the pureness guarantee is essential to ensuring consistency of clients’ global models. It is imperative that the global update function is just that, a function, determined uniquely by its inputs.

### **8.2.2 RQ2. What are the measurable differences in course engagement found between the single player and multiplayer games?**

Students making multiplayer games for their projects were also more likely to report that they were happy with their decision to do so compared to those who did single player (Figure 7.11). Multiplayer game builders also reported a greater degree of project enjoyment (Figure 7.12). This is consistent with data from the online IDE, which showed a slightly higher mean and median number of compiles, a much larger number of modules created, and a slightly higher number of lines of code per compile for those who chose multiplayer games. Whereas the average number of compiles was actually lower in the second semester for single player game creators, it went up in the second semester for those who made multiplayer games.

The “time fairness” metric did not show any significant difference, especially in the median. Anecdotally, some students really took to the multiplayer game framework, spending a lot of time refining their games, making the mean slightly higher. This metric overall does not tell a strong story other than that work (at least in terms of compiles) is highly-concentrated (likely on lab and due dates) and that future work should look at how to encourage students to spread out their work more. We saw very similar time fairness metrics when analyzing the compile data of Capstone

students [56]. More work should be done to see if this fairness metric behaves well or conveys much information with so many data points.

### **8.2.3 RQ3. What evidence is there that this approach successfully provides a LFHC environment for students?**

There was evidence that the multiplayer framework allowed for a higher ceiling, keeping students more engaged with the course according to the metrics of code compilations in the WebIDE. The results of the survey indicates that students who chose to use TEASync had more prior programming experience than those who did not, suggesting it was an outlet for those who wished to push into new territory.

While some students were able to create complex multiplayer games, it is clear there are still some students who found the framework to be too complex or at least intimidating. This was obvious from some of the comments in the focus group, where one student even stated outright that it “favours students who have had a bigger background in computer science.” Even students who chose to do it for their project had the difficulty of a lack of documentation precluding them from using advanced features. It was also clear from the survey results that students who chose to make multiplayer games were more likely to report more experience in programming prior to university (Figure 7.13), further adding to the evidence that the framework was perceived as too difficult for beginner programmers.

The lower floor was provided by the single player option and especially the state diagram tool, which allowed easy app design and code generation. A future work item is to explore how to further lower the floor of TEASync.

## 8.3 Future Work

There are several areas of future work, both technically and pedagogically.

### 8.3.1 TEASync/LG-MVU Improvements

Feedback from students and anecdotal experiences in the classroom point to the need of a more powerful model for certain use cases, especially games with simulations. A global time tick would be useful for such cases, but would present its own performance and network traffic challenges.

Additionally, the current split into the simplified and advanced APIs was not a good split. Applications requiring communication between the local and global updates motivate the need for an API in between these two, which does not need to introduce the full complexity of Elm commands to achieve message brokering between the local and global.

### 8.3.2 Formalized Algebra for LG-MVU

Future work in formalizing the various concurrency schemes and analyzing which properties of the global update function and messages lead to race conditions would be an interesting future study. Section 4.5 gave an overview of the desirable properties of global models and messages, but these could be formalized using a system like Agda, and then properties could be proven to hold.

While the students were able to make applications without considering the properties listed in Section 4.5, as their programs became more complex, they would be more likely to run into race conditions. This points to TEASync as a teaching tool for

more than just concurrency but as a real-world example of where discrete mathematics concepts are important to ensure desirable properties. It would be advantageous to explore the creation of an algebra of messages to create a formal model of TEASync, drawing upon existing research in concurrency and event-driven programming. This would create a framework for proving that a TEASync program cannot have race conditions, and for removing them when found. Assignment/test questions in such a course could centre around designing global message and update functions that meet certain properties, and proving that they do.

### **8.3.3 Experiments for Concurrency Schemes**

Another future work item would be to empirically experiment with the different concurrency schemes, to determine their performance characteristics. In particular, it would be worth experimenting with the parameters in the message-based synchronization with distributed folding scheme to determine good parameters.

### **8.3.4 Visual Representation of Concurrency Model**

Following the success of the state diagram tool, current and future work is aimed at creating a visual tool to create TEASync applications. We believe this would make concurrent applications even more appealing to new computer science students. One idea is to use state charts to expand upon the current LG-MVU architecture, allowing things like global timers to be included naturally where right now it does not fit into the model. This would allow the programmer to abstract beyond the single local and single global update to organize their programs in more sophisticated ways.

### **8.3.5 Study Security Implications**

In the current work, the security of the TEASync framework is not a focus. However, given the distributed nature of the framework, security is likely to be an issue. Exploring distributed data access schemes is a potential for future research.

### **8.3.6 Camps for K-12 Education**

Given the success of other elements of our coding outreach program, we hope to create activities for K-12 education including multiplayer elements. These could include summer camps and in-class curricula. We believe student engagement would benefit from multiplayer games in the K-12 space.

# Appendix A

## Usability Study Instruments

This Appendix contains the raw usability study instruments, including the pre- and post-implementation surveys (Appendices A.1 and A.2) and the questions in the focus group script (Appendix A.3).

### A.1 Pre-Implementation Survey Questions

The pre-survey questions are shown in Figures A.1, A.2, A.3, A.4, A.5, A.6, A.7, and A.8.

### A.2 Post-Implementation Survey Questions

The post-survey questions are shown in Figures A.9, A.10, A.11, A.12, and A.13.

## CompSci 1XD3 Pre-Survey

Hi, Christopher. When you submit this form, the owner will see your name and email address.

### Section 1: Informed Consent

You can find the study's letter of information and consent here: [https://mcmasteru365-my.sharepoint.com/:b/g/personal/schankuc\\_mcmaster\\_ca/Ef1rdYwNQFNlvEuWstCZF148qkzd9FI3w3z36ydnb8Jd7g?e=Rdlqnl](https://mcmasteru365-my.sharepoint.com/:b/g/personal/schankuc_mcmaster_ca/Ef1rdYwNQFNlvEuWstCZF148qkzd9FI3w3z36ydnb8Jd7g?e=Rdlqnl). Please read it and then complete the digital consent signature below.

**CONSENT**

- I have read the information presented in the information letter about a study being conducted by Christopher Schankula, Dr. Christopher Anand, and Dr. Spencer Smith, of McMaster University.
- I have had the opportunity to ask questions about my involvement in this study and to receive additional details I requested.
- I understand that if I agree to participate in this study, I may withdraw from the study at any time or up until **May 1st, 2024**.

1. If you agree with the above bullet points, please type your full name in this box. This will be treated as a digital signature of the letter of information and consent linked above.

Enter your answer

2. Do you agree for your [STaBL.Rocks](#) statistics to be included in the dataset for analysis and summary reporting, as described in the Letter of Information? Statistics will be reported as summary information and any one person's individual statistics will **not** be reported.

☐ Yes

☐ No

3. Would you like a copy of the study results? If yes, we will send them to your McMaster email address.

☐ Yes

☐ No

4. Would you like to be entered in a draw for the gift card?

☐ Yes

☐ No

5. Do you agree to allow your anonymized survey data to be stored and used for future research as described in the Letter of Information?

☐ Yes

☐ No

6. Do you agree to be added to a recruitment mailing to be contacted for future research? If so, we will email your McMaster email with any future research opportunities.

☐ Yes

☐ No

Figure A.1: Section 1 of the pre-survey, gathering informed consent and demographic data.



### CompSci 1XD3 Pre-Survey

#### Section 2: Demographics and Basic Information

7. What program are you currently enrolled in? (Note: Due to the low enrolment in CompSci 1XD3 of students in programs other than Computer Science, and Math & Computer Science, we are not asking you to specify your program if "Other").

☐ Computer Science

☐ Math & Computer Science

☐ Other

8. What is your current year of study?

☐ 1st year

☐ 2nd year

☐ 3rd year

☐ 4th year

☐ 5th year or more

9. Which of the following do you identify as? If you would like to self-identify as another gender not listed here, please enter it into the "Other" option.

☐ Female

☐ Male

☐ Non-binary

☐ Prefer not to answer

☐ Other

10. How much programming experience did you have prior to entering university?

☐ None

☐ Less than 1 month

☐ 1-3 months

☐ 3-6 months

☐ 6-12 months

☐ 1 year

☐ 2 years

☐ 3 years or more

Figure A.2: Section 2 of the pre-survey, gathering data about past experience with game development.

11. How many computer science courses did you take in high school? ☐

☐ 0

☐ 1

☐ 2

☐ 3

☐ 4 or more

12. How often do you work on your own programming projects (outside of school)? ☐

☐ Every day

☐ 2-3 times per week

☐ Once per week

☐ 2-3 times per month

☐ Once per month or less

☐ Never

13. How many total lines of code is the largest programming project (either at school or at work) you have completed in a group? ☐

☐ I have not worked on a group programming project.

☐ 1-100 lines of code

☐ 100-500 lines of code

☐ 500-1000 lines of code

☐ 1000+ lines of code

14. How often do you play each of the following types of video games? ☐

	Every day	2-3 times per week	Once per week	2-3 times per month	Once per month or less	Never
Single player	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Multiplayer	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

15. Please rate how much you enjoy playing the following types of video games, with 1 being strongly dislike and 5 being extremely enjoyable. ☐

	1 - Extremely unenjoyable	2 - Unenjoyable	3 - Neither enjoyable nor unenjoyable	4 - Enjoyable	5 - Extremely enjoyable
Single player	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Multiplayer	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Figure A.3: Section 2 of the pre-survey (continued), gathering data about past experience with game development.

CompSci 1XD3 Pre-Survey

Section 3 - YN

17. You indicated that you **have** created a **single player** game before. For each of the following aspects, rate the difficulty you experienced while creating your **single player** game.

	Very Difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

18. You indicated that you have **not** created a **multiplayer** game before. For each of the following aspects, rate how difficult you *believe* each of the following would be while creating a **multiplayer** game.

	Very Difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Back

Next

Figure A.4: Section 3 of the pre-survey, gathering information about past experiences with game development. (Single player: yes, Multiplayer: no)

CompSci 1XD3 Pre-Survey

Section 3 - NY

17. You indicated that you have **not** created a **single player** game before. For each of the following aspects, rate how difficult you *believe* each of the following would be while creating a **single player** game.

	Very Difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

18. You indicated that you **have** created a **multiplayer** game before. For each of the following aspects, rate the difficulty you experienced while creating your **multiplayer** game.

	Extremely Difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Back

Next

Figure A.5: Section 3 of the pre-survey, gathering information about past experiences with game development. (Single player: no, Multiplayer: yes)

CompSci 1XD3 Pre-Survey

Section 3 - YY

17. You indicated that you *have* created a **single player** game before. For each of the following aspects, rate the difficulty you experienced while creating your **single player** game.

	Very Difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

18. You indicated that you *have* created a **multiplayer** game before. For each of the following aspects, rate the difficulty you experienced while creating your **multiplayer** game.

	Very Difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Back

Next

Figure A.6: Section 3 of the pre-survey, gathering information about past experiences with game development. (Single player: no, Multiplayer: yes)

CompSci 1XD3 Pre-Survey

Section 3 - NN

17. You indicated that you have **not** created a **single player** game before. For each of the following aspects, rate how difficult you *believe* each of the following would be while creating a **single player** game.

	Very Difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

18. You indicated that you have **not** created a **multiplayer** game before. For each of the following aspects, rate how difficult you *believe* each of the following would be while creating a **multiplayer** game.

	Very Difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Back

Next

Figure A.7: Section 3 of the pre-survey, gathering information about past experiences with game development. (Single player: no, Multiplayer: no)

CompSci 1XD3 Pre-Survey

Section 4 - Final Question

19. For each of the following statements, please read the statement carefully and rate your level of agreement with the statement.

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
I don't feel that I understand what Design Thinking is	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
It is important for software professionals to understand user needs	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Computer science and coding are forces that can be used for evil	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I prefer to code when it has a bigger purpose	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
It is difficult to know what my user's needs are	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I enjoy programming	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I enjoy programming in Elm	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I am excited about the computer science program	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I wouldn't like to pursue a career in computing	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

20. Do you have any other questions or comments?

Enter your answer

Back

Submit

Figure A.8: Section 4 of the pre-survey, gathering a baseline of preferences about aspects of computer science.

### CompSci 1XD3 Post-Survey

Hi, Christopher. When you submit this form, the owner will see your name and email address.

#### Section 1: Informed Consent

You can find the study's letter of information and consent here: <will insert link>. Please read it and then complete the digital consent signature below.

**CONSENT**

- I have read the information presented in the information letter about a study being conducted by Christopher Schankula, Dr. Christopher Anand, and Dr. Spencer Smith, of McMaster University.
- I have had the opportunity to ask questions about my involvement in this study and to receive additional details I requested.
- I understand that if I agree to participate in this study, I may withdraw from the study at any time or up until **May 1st, 2024**.

1. If you agree with the above bullet points, please type your full name in this box. This will be treated as a digital signature of the letter of information and consent linked above.

Enter your answer

2. Do you agree for your [STaBL.Rocks](#) statistics to be included in the dataset for analysis and summary reporting, as described in the Letter of Information? Statistics will be reported as summary information and any one person's individual statistics will **not** be reported.

☐ Yes

☐ No

3. Would you like a copy of the study results? If yes, we will send them to your McMaster email address.

☐ Yes

☐ No

4. Would you like to be entered in a draw for the gift card?

☐ Yes

☐ No

5. Do you agree to allow your anonymized survey data to be stored and used for future research as described in the Letter of Information?

☐ Yes

☐ No

6. Do you agree to be added to a recruitment mailing list to be contacted for future research? If so, we will email your McMaster email with any future research opportunities.

☐ Yes

☐ No

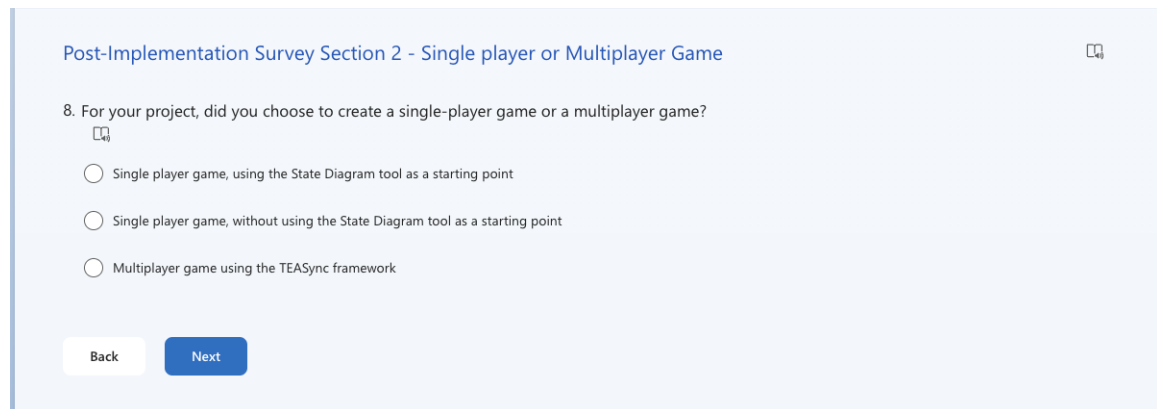
7. Did you **not** complete the pre-implementation survey earlier this year but want to complete the relevant questions now? Answering "Yes" will take you through the pre-implementation survey questions and then through the post-implementation survey questions. You will receive 2 entries in the draw for doing both. If you have already completed the pre-implementation survey, please answer "No".

☐ Yes

☐ No

Figure A.9: Section 1 of the post-survey, gathering informed consent and demographic data, and asking whether they would like to answer the pre-survey questions if they had not already done so.





Post-Implementation Survey Section 2 - Single player or Multiplayer Game

8. For your project, did you choose to create a single-player game or a multiplayer game?

☐ Single player game, using the State Diagram tool as a starting point

☐ Single player game, without using the State Diagram tool as a starting point

☐ Multiplayer game using the TEASync framework

Back Next

Figure A.10: Section 2 of the post-survey, asking which type of game the student made. This will determine whether they are asked about their experience making a single player or multiplayer game (Figure A.11 or A.12, respectively).

## A.3 Focus Group Scripted Questions

Below is the list of scripted questions for the focus group. These questions were the original questions in the script, but the script allowed for follow-up questions to be asked if needed.

- What parts of the project did you enjoy? Why?
- For those who chose to make a single player game, what reasons did your group have for choosing to do so?
  - Follow-up: were you happy with your decision to choose this? Why or why not?
- For those who chose to make a multiplayer game, what reasons did your group have for choosing to do so?
  - Follow-up: were you happy with your decision to choose this? Why or why not?

Section 2 - SP

9. Of the following options, which one best describes how you formed your group?

☐ I chose my group first, then we decided to make a single player game.
   
☐ I wanted to make a single player game and chose my group based on that

10. You indicated that you chose to make a **single player** game for your project. Thinking about your project this semester, please rate how much each of the following statements applies to you and your group.

	Does not apply to us	Somewhat applies to us	Applies to us	Completely applies to us
We enjoy playing single player games, so we wanted to make one	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought it would be more interesting to create a single player game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought the resulting game would be more engaging being single player	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought we could create a better project by making a single player game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought we could get a better mark by creating this kind of game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We had more experience making single player games	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We wanted to try something new	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought a single player game would be more applicable to the Design Thinking problem	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

11. Thinking about the **single player** game your team created, please rate the level of difficulty you experienced with each of the following aspects.

	Extremely difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Back
Next

Figure A.11: Section 2 of the post-survey, for students who chose a single-player game.

Section 2 - MP

9. Of the following options, which one best describes how you formed your group?

☐ I chose my group first, then we decided to make a multiplayer game.
   
☐ I wanted to make a multiplayer game and chose my group based on that

10. You indicated that you chose to make a **multiplayer** game for your project. Thinking about your project this semester, please rate how much each of the following statements applies to you and your group.

	Does not apply to us	Somewhat applies to us	Applies to us	Completely applies to us
We enjoy playing multiplayer games, so we wanted to make one	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought it would be more interesting to create a multiplayer game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought the resulting game would be more engaging being multiplayer	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought we could create a better project by making a multiplayer game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought we could get a better mark by creating this kind of game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We had more experience making multiplayer games	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We wanted to try something new	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
We thought a multiplayer game would be more applicable to the Design Thinking problem	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

11. Thinking about the **multiplayer** game your team created, please rate the level of difficulty you experienced with each of the following aspects.

	Extremely difficult	Difficult	Neither difficult nor easy	Easy	Extremely easy
Designing game mechanics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Designing graphics	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Writing the storyline/script	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Programming/coding the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Learning the tools needed to make the game	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Back
 Next

Figure A.12: Section 2 of the post-survey, for students who chose a multiplayer game.

Section 3

12. For each of the following statements, rate your agreement with the statement.

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
I feel everyone in my team contributed equally to the project	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The State Diagram is a useful tool for creating single player games	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The State Diagram tool was difficult to learn	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The State Diagram tool was powerful enough to do what we wanted to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The TEASync framework is a useful tool for creating multiplayer games	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The TEASync framework was difficult to learn	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The TEASync framework was powerful enough to do what we wanted to do	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The <a href="#">STaBL Rocks</a> platform made it easy for us to collaborate	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The Design Thinking components of the project were more difficult than the technical (coding) aspects of the project	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Our group is happy with our decision to make the type of game we made (single player or multiplayer)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

13. Compared to other course projects, how much did you enjoy the 1XD3 project?

	-2 - Did not enjoy at all	-1 - Did not enjoy	0 - Indifferent	1 - Enjoyed	2 - Enjoyed very much
Project enjoyment	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

14. Do you have any other questions or comments?

Enter your answer

Back
Submit

Figure A.13: Section 3 of the post-survey, asking agreement with several statements about the course and tools.

- Which aspects of the project were difficult for you?
  - Follow up questions if they don't mention it explicitly:
    - \* Which parts of the design thinking process were difficult?
    - \* Which parts of the technical aspects were difficult?
- Are you satisfied with the creativity and number of the ideas your group came up with? Did it seem the questions belonged to the group or to the person who came up with the idea?
- How did using the “Game Matrix” compare idea generation methods you used in the past, including “Brainstorming”?
- How do you describe your overall experience with the Design Thinking worksheet? What worked well? What could be improved?
- Do you have suggestions to improve the project next year?
- Do you have suggestions to improve the tools we used in class (e.g., State Diagram creator, STaBL.Rocks coding system, TEASync multiplayer framework)?
- How can you use what you learned in the project in your future career in the computer science field or otherwise?
- Are there any other comments or questions anyone else would like to make known?

# Appendix B

## Code Examples

### Example B.0.1: TEASync Counting Example Code

```
myShapes localModel globalModel =  
  [  
    text (String.fromInt globalModel.count)  
      |> centered  
      |> filled black  
      |> move (0, -3)  
    , button green "+" (localModel.mousingOver == Just ←  
      Increment)  
      |> move (0, 30)  
      |> notifyTap (GlobalMsg <| Crement Increment)  
      |> notifyEnter (LocalMsg <| MouseOver Increment)  
      |> notifyLeave (LocalMsg MouseOff)  
    , button red "-" (localModel.mousingOver == Just ←  
      Decrement)
```

```

        |> move (0, -30)
        |> notifyTap (GlobalMsg <| Crement Decrement)
        |> notifyEnter (LocalMsg <| MouseOver Decrement)
        |> notifyLeave (LocalMsg MouseOff)
    ]

button colour txt outlined =
    group
    [
        circle 10
        |> filled colour
        |> (if outlined then addOutline (solid 1) black ←
            else identity)
        , text txt
        |> centered
        |> filled black
        |> move (0, -3)
    ]

localUpdate : LocalMsg -> LocalModel -> GlobalModel -> ←
    LocalModel

localUpdate msg localModel globalModel =
    case msg of
        Tick t _ -> { localModel | time = t }

```

```
    MouseOver crement -> { localModel | mousingOver = ←  
      Just crement }  
    MouseOff -> { localModel | mousingOver = Nothing }  
  
globalUpdate : GlobalMsg -> GlobalModel -> GlobalModel  
globalUpdate msg globalModel =  
  case msg of  
    Crement Increment -> { globalModel | count = ←  
      globalModel.count + 1 }  
    Crement Decrement -> { globalModel | count = ←  
      globalModel.count - 1 }  
  
initLocal : LocalModel  
initLocal = { time = 0, mousingOver = Nothing }  
  
initGlobal : GlobalModel  
initGlobal = { count = 0 }
```

#### Example B.0.2: Pong Code Example

```
myShapes localModel globalModel =  
  [  
    case localModel.localState of  
      AtMainMenu ->  
        let
```



```
        player = case globalModel.gameState of
                    PlayerOneJoined -> PlayerTwo
                    _ -> case localModel.localState of
                            PlayingGame _ p -> p
                            _ -> PlayerOne

in
  group
    [
      text "Welcome!"
      |> centered
      |> fixedwidth
      |> filled black
      |> move(0, 20)
    , group
      [
        roundedRect 50 20 5 |> filled black
        , text "Play"
        |> centered
        |> fixedwidth
        |> filled white
        |> move (0, -4)
      ]
      |> notifyTap (LocalMsg <| LocalPlayerJoin↔
                    player)
    ]
```

```
InLobby player ->
  group
  [
    text ("You are " ++ (if player == PlayerOne <-
      then "Player 1" else "Player 2") ++ ".")
    |> centered
    |> fixedwidth
    |> size 6
    |> filled black
    |> move (0, 20)
    , text (if globalModel.gameState == <-
      PlayerOneJoined then "Waiting for player 2."<-
      else "Both players joined.")
    |> centered
    |> fixedwidth
    |> size 6
    |> filled black
    |> move (0, 10)
    , group
    [
      roundedRect 50 20 5 |> filled (if <-
        globalModel.gameState /= <-
        PlayerTwoJoined then grey else black<-
      )
      , text "Start"
```

```
        |> centered
        |> fixedwidth
        |> filled white
        |> move (0, -4)
    ]
    |> move (0, -20)
    |> (if globalModel.gameState == ↵
        PlayerTwoJoined then notifyTap (↵
            GlobalMsg StartGame) else identity)
    ]
PlayingGame localGameState player ->
let
    (player1Pos, player2Pos) = globalModel.↵
        playerPos
    (player1Score, player2Score) = globalModel.↵
        score
    ballPos = localGameState.pos
in
group
[
    rect 192 128
        |> filled black
        |> notifyMouseMoveAt (\(x,y) -> GlobalMsg ↵
            <| MovePlayer player y )
```

```
        |> notifyTouchMoveAt ( \(x,y) -> GlobalMsg ←  
            <| MovePlayer player y )  
    , line (0, 64) (0, -64)  
        |> outlined (dotted 2) white  
    , rect 2 20  
        |> filled white  
        |> move (-90, player1Pos)  
    , rect 2 20  
        |> filled white  
        |> move (90, player2Pos)  
    , square 3  
        |> filled white  
        |> move ballPos  
    , text (String.fromInt player1Score)  
        |> fixedwidth  
        |> centered  
        |> size 14  
        |> filled white  
        |> move (-80, 50)  
    , text (String.fromInt player2Score)  
        |> fixedwidth  
        |> centered  
        |> size 14  
        |> filled white  
        |> move (80, 50)
```

```

    ]

]

-- Your update function goes here
localUpdate : LocalMsg -> LocalModel -> GlobalModel -> ( ↵
    LocalModel, Cmd LocalMsg, Cmd GlobalMsg )
localUpdate msg localModel globalModel =
    case msg of
        Tick t _ ->
            case localModel.localState of
                PlayingGame localGameState player ->
                    let
                        (x,y) = localGameState.pos
                        (dx,dy) = localGameState.vel
                        (p1y, p2y) = globalModel.playerPos
                        (ndx, ndy, cmd) =
                            case player of
                                PlayerOne ->
                                    if dx <= 0 && x <= -87 && x >= -90 && ↵
                                        p1y - 11.5 <= y && y <= p1y + ↵
                                            11.5 then
                                                let
                                                    vel = sqrt (dx*dx + dy*dy) * 1.1
                                                    angle = (y- p1y) * 7

```

```

        (newdX, newdY) = (vel * cos (←
            degrees angle), vel * sin (←
            degrees angle))

    in

    (newdX, newdY, newMsg <| HitBall ←
        player (x,y) (newdX, newdY))
else
    (dx, dy, Cmd.none)
PlayerTwo ->
    if dx >= 0 && x >= 87 && x <= 90 && ←
        p2y - 11.5 <= y && y <= p2y + 11.5 ←
        then
        let
            vel = sqrt (dx*dx + dy*dy) * 1.1
            angle = (p2y - y) * 7 + 180
            (newdX, newdY) = (vel * cos (←
                degrees angle), vel * sin (←
                degrees angle))

        in

        (newdX, newdY, newMsg <| HitBall ←
            player (x,y) (newdX, newdY))
    else
        (dx, dy, Cmd.none)
(fdx, fdy) = if y > 62.5 || y < -62.5 then ←
    (ndx, -ndy) else (ndx, ndy)

```

```

        scored = if x >= 100 then Just PlayerOne
                  else if x <= -100 && player == ←
                        PlayerTwo then Just PlayerTwo
                  else Nothing
in
  ( {localModel |
    localState =
      PlayingGame
      { localGameState |
        pos = case scored of
              Nothing -> (x+fdx, y+fdy←
                          )
              Just PlayerOne -> (10, ←
                                0)
              Just PlayerTwo -> (-10, ←
                                0)
        , vel = case scored of
              Nothing -> (fdx, fdy)
              Just PlayerOne -> ←
                (12/30, 0)
              Just PlayerTwo -> ←
                (-12/30, 0)
        }
      player }
    , Cmd.none

```

```
    , Cmd.batch
    [
        cmd
    , case scored of
        Just p -> if player == p then
            Cmd.batch
            [
                newMsg <| ↵
                    PlayerScored p
            , newMsg <| HitBall↵
                player
            (case scored of
                Nothing -> ↵
                    (x+fdx, ↵
                        y+fdy)
                Just ↵
                    PlayerOne↵
                    -> (10,↵
                        0)
                Just ↵
                    PlayerTwo↵
                    -> ↵
                    (-10, 0)↵
                    )
            (case scored of
```



```

Nothing -> ↵
    (fdx, ↵
        fdy)
Just ↵
    PlayerOne↵
        -> ↵
            (12/30, ↵
                0)
Just ↵
    PlayerTwo↵
        -> ↵
            (-12/30,↵
                0))

    ]

    else Cmd.none

    Nothing -> Cmd.none

    ]

    )

    _ -> ( localModel , Cmd.none , Cmd.none )

LocalPlayerJoin player ->

    ( { localModel | localState = InLobby player }

    , Cmd.none

    , newMsg <| PlayerJoin player

    )

LocalStart ->

```

```

let
  player =
    case localModel.localState of
      InLobby p -> p
      PlayingGame _ p -> p
      _ -> PlayerOne
in
  ( { localModel | localState =
      case localModel.localState of
        AtMainMenu -> localModel.localState
        _ -> PlayingGame { pos = (0,0), vel = ↵
          (-12/30,0) } player }
    , Cmd.none
    , Cmd.none
  )
UpdatePosition player (x, y) (dx, dy) ->
  case localModel.localState of
    PlayingGame localGameState p ->
      if p /= player then
        ( {localModel | localState = PlayingGame { ↵
          localGameState | pos = (x, y), vel = (dx↵
            , dy) } p }
          , Cmd.none
          , Cmd.none
        )

```

```

        else
            (localModel, Cmd.none, Cmd.none)
        _ -> ( localModel, Cmd.none, Cmd.none )

globalUpdate : GlobalMsg -> GlobalModel -> ( GlobalModel, Cmd GlobalMsg, Cmd LocalMsg )
globalUpdate msg globalModel =
    case msg of
        MovePlayer player pos ->
            ( { globalModel |
                playerPos =
                    if player == PlayerOne then
                        Tuple.mapFirst (\_ -> pos) <-
                            globalModel.playerPos
                    else
                        Tuple.mapSecond (\_ -> pos) <-
                            globalModel.playerPos
            }
            , Cmd.none
            , Cmd.none
            )
        HitBall player pos vel ->
            ( globalModel
            , Cmd.none
            , newMsg <| UpdatePosition player pos vel

```

```
)  
  
PlayerJoin player ->  
  ( { globalModel |  
      gameState =  
        case player of  
          PlayerOne -> PlayerOneJoined  
          PlayerTwo -> PlayerTwoJoined  
      }, Cmd.none, Cmd.none )  
  
StartGame ->  
  ( { globalModel |  
      gameState = Playing  
      }, Cmd.none  
      , newMsg LocalStart  
      )  
  
PlayerScored player ->  
  ( { globalModel |  
      score =  
        if player == PlayerOne then  
          Tuple.mapFirst (\n -> n + 1)  $\hookleftarrow$   
            globalModel.score  
        else  
          Tuple.mapSecond (\n -> n + 1)  $\hookleftarrow$   
            globalModel.score  
      }  
      , Cmd.none
```

```
        , Cmd.none
      )

-- Your initial model goes here
initLocal : (LocalModel, Cmd LocalMsg)
initLocal = ({ time = 0, localState = AtMainMenu, ↵
              lastTime = 0 }, Cmd.none)

initGlobal : (GlobalModel, Cmd GlobalMsg)
initGlobal = ({ gameState = NoPlayersJoined, score = (0, ↵
              0), playerPos = (0, 0) }, Cmd.none)

-- Your local subscriptions go here
localSubscriptions : LocalModel -> GlobalModel -> Sub ↵
                  LocalMsg
localSubscriptions localModel globalModel = Sub.none

-- Your global subscriptions go here
globalSubscriptions : LocalModel -> GlobalModel -> Sub ↵
                   GlobalMsg
globalSubscriptions localModel globalModel = Sub.none

appConfig =
  { initLocal = \_ -> initLocal
```

```
, initGlobal = \_ -> initGlobal
, localUpdate = localUpdate
, globalUpdate = globalUpdate
, view = view
, localSubscriptions = localSubscriptions
, globalSubscriptions = globalSubscriptions
, codecGlobalModel = JSON Codec.Encoders.<-
    encodeGlobalModel Codec.Decoders.decodeGlobalModel
, codecGlobalMsg = JSON Codec.Encoders.<-
    encodeGlobalMsg Codec.Decoders.decodeGlobalMsg
}

-- Your main function goes here
main : TEASyncGSVGAppWithTick () LocalModel GlobalModel <-
    LocalMsg GlobalMsg
main =
    teaSyncAppWithTick Tick
    appConfig

-- You view function goes here
view : LocalModel -> GlobalModel -> { title: String, body<-
    : Collage (TEASync.Msg LocalMsg GlobalMsg GlobalModel<-
    ) }
view localModel globalModel =
    {
```

```
    title = "My App Title"  
    , body = collage 192 128 (myShapes localModel ↪  
      globalModel)  
  }
```

# Bibliography

- [1] C. Alphonse and P. Ventura. Using graphics to support the teaching of fundamental object-oriented principles in cs1. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 156–161, 2003.
- [2] A. P. Ambrósio, F. M. Costa, L. Almeida, A. Franco, and J. Macedo. Identifying cognitive abilities to improve cs1 outcome. In *2011 Frontiers in Education Conference (FIE)*, pages F3G–1. IEEE, 2011.
- [3] C. K. Anand, G. Dulai, L. Yao, M. Arief, O. D’Mello, S. S. Menon, and C. W. Schankula. *Creating with Code*. McMaster University, 2023. ISBN 978-1-7388695-0-3. URL <https://macsphere.mcmaster.ca/handle/11375/28334>.
- [4] C. Bachmann, A. Maximova, T. Kohn, and D. Komm. Webtigerpython—a low-floor high-ceiling python ide for the browser. *arXiv preprint arXiv:2410.07001*, 2024.
- [5] J. C. Blake-West and M. U. Bers. Scratchjr design in practice: Low floor, high ceiling. *International Journal of Child-Computer Interaction*, 37:100601, 2023.



- [6] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1(2):147–164, 1988.
- [7] P. Buiras and A. Russo. Lazy programs leak secrets. In *Nordic Conference on Secure IT Systems*, pages 116–122. Springer, 2013.
- [8] M. M. Chakravarty and G. Keller. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming*, 14(1):113–123, 2004.
- [9] H. B. Christensen and M. E. Caspersen. Frameworks in cs1: a different way of introducing event-driven programming. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 75–79, 2002.
- [10] E. Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 30, 2012.
- [11] E. Czaplicki. A farewell to frp. <https://elm-lang.org/news/farewell-to-frp>, 2016. Accessed: 2024-05-09.
- [12] E. Czaplicki. Status update (3 nov 2021). <https://discourse.elm-lang.org/t/status-update-3-nov-2021/7870>, 2021. Accessed: 2024-09-06.
- [13] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189, 2002.

- [14] C. d’Alves, T. Bouman, C. Schankula, J. Hogg, L. Noronha, E. Horsman, R. Siddiqui, and C. K. Anand. Using elm to introduce algebraic thinking to k-8 students. *arXiv preprint arXiv:1805.05125*, 2018.
- [15] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. *ACM SIGPLAN Notices*, 48(6): 321–332, 2013.
- [16] digitally induced GmbH. Ihp: Integrated haskell platform. <https://ihp.digitallyinduced.com/>. Accessed: 2024-05-31.
- [17] V. Dolgopolas, T. Jevsikova, and V. Dagiene. From android games to coding in c—an approach to motivate novice engineering students to learn programming: A case study. *Computer Applications in Engineering Education*, 26(1):75–90, 2018.
- [18] J. Dunfield and N. R. Krishnaswami. Complete and easy bidirectional type-checking for higher-rank polymorphism. *ACM SIGPLAN Notices*, 48(9):429–442, 2013.
- [19] M. S. El-Nasr and B. K. Smith. Learning through game modding. *Computers in Entertainment (CIE)*, 4(1):7–es, 2006.
- [20] I. Fette and A. Melnikov. The WebSocket Protocol. Internet Engineering Task Force, December 2011. URL <https://www.rfc-editor.org/rfc/rfc6455.html>. RFC 6455.
- [21] G. Gadanidis, S. Floyd, J. Hughes, I. Namukasa, and R. Scucuglia. Coding in

- the ontario mathematics curriculum, 1–8: Might it be transformational. *Journal of Computers in Mathematics and Science Teaching*, 40:357–373, 2021.
- [22] P. Gestwicki and F.-S. Sun. Teaching design patterns through computer game development. *Journal on Educational Resources in Computing (JERIC)*, 8(1): 1–22, 2008.
- [23] M. H. Goldwasser and D. Letscher. A graphics package for the first day and beyond. *ACM SIGCSE Bulletin*, 41(1):206–210, 2009.
- [24] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM systems journal*, 45(3):451–461, 2006.
- [25] Haskell Language. Haskell language - an advanced, purely functional programming language, 2024. URL <https://www.haskell.org/>. Accessed: 2024-05-31.
- [26] C. Hill, H. A. Dwyer, T. Martinez, D. Harlow, and D. Franklin. Floors and flexibility: Designing a programming environment for 4th-6th grade classrooms. In *Proceedings of the 46th ACM technical Symposium on computer science education*, pages 546–551, 2015.
- [27] R. Hindley et al. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.
- [28] Z. Hu, J. Hughes, and M. Wang. How functional programming mattered. *National Science Review*, 2(3):349–370, 2015.
- [29] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

- [30] P. Hudak. *The Haskell school of expression: learning functional programming through multimedia*. Cambridge University Press, 2000.
- [31] J. Hughes. Why functional programming matters. *The computer journal*, 32(2): 98–107, 1989.
- [32] International Organization for Standardization. Information technology – syntactic metalanguage – extended bnf. Standard ISO/IEC 14977:1996, International Organization for Standardization, Geneva, CH, 1996.
- [33] C. Jaspan and C. Sadowski. *No Single Metric Captures Productivity*, pages 13–20. Apress, Berkeley, CA, 2019. ISBN 978-1-4842-4221-6. doi: 10.1007/978-1-4842-4221-6\_2. URL [https://doi.org/10.1007/978-1-4842-4221-6\\_2](https://doi.org/10.1007/978-1-4842-4221-6_2).
- [34] S. Joosten, K. Van Den Berg, and G. Van Der Hoeven. Teaching functional programming to first-year students. *Journal of Functional Programming*, 3(1): 49–65, 1993.
- [35] C. Kieran. Algebraic thinking in the early grades: What is it. *The mathematics educator*, 8(1):139–151, 2004.
- [36] D. Leyzberg and C. Moretti. Teaching cs to cs teachers: Addressing the need for advanced content in k-12 professional development. In *Proceedings of the 2017 ACM SIGCSE technical symposium on Computer Science Education*, pages 369–374, 2017.
- [37] D. Liben-Nowell and A. N. Rafferty. Student motivations and goals for cs1:

- themes and variations. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*, pages 237–243, 2022.
- [38] A. Lukkarinen, L. Malmi, and L. Haaranen. Event-driven programming in programming education: a mapping review. *ACM Transactions on Computing Education (TOCE)*, 21(1):1–31, 2021.
- [39] F. Maiorana. Interdisciplinary computing for ste (a) m: a low floor high ceiling curriculum. *Innovations, Technologies and Research in Education*, 37, 2019.
- [40] G. Melfe, A. Fonseca, and J. P. Fernandes. Evaluation of the impact on energy consumption of lazy versus strict evaluation of haskell data-structures. In *Proceedings of the XXII Brazilian Symposium on Programming Languages*, pages 83–89, 2018.
- [41] Merriam-Webster. Codec. <https://www.merriam-webster.com/dictionary/codec>, 2024. Accessed: 2024-05-17.
- [42] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
- [43] A. Mohamed. Designing a cs1 programming course for a mixed-ability class. In *Proceedings of the western Canadian conference on computing education*, pages 1–6, 2019.
- [44] J. G. Morris. Notes on hindley-milner polymorphism. <https://jgbm.github.io/eecs662f17/Notes-on-HM.html>, 2017. Accessed: 2024-05-17.
- [45] T. U. of Glasgow. `Control.Concurrent.STM.TQueue`.

- <https://hackage.haskell.org/package/stm-2.5.3.1/docs/Control-Concurrent-STM-TQueue.html>, 2010. Accessed: 2024-05-31.
- [46] D. Oleynikov and G. Dreimanis. History of the haskell programming language. <https://serokell.io/blog/haskell-history>, 2019. Accessed: 2024-05-31.
- [47] S. A. Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic books, 2020.
- [48] P. Pasupathi, C. W. Schankula, N. DiVincenzo, S. Coker, and C. K. Anand. Teaching interaction using state diagrams. *arXiv preprint arXiv:2207.12701*, 2022.
- [49] L. Payr. *Refinement types for Elm*. PhD thesis, Universität Linz, 2021.
- [50] R. Pecinovský, J. Pavlíčková, and L. Pavlíček. Let’s modify the objects-first approach into design-patterns-first. In *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, pages 188–192, 2006.
- [51] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of json schema. In *Proceedings of the 25th international conference on World Wide Web*, pages 263–273, 2016.
- [52] M. Resnick. *Lifelong kindergarten: Cultivating creativity through projects, passion, peers, and play*. Mit Press, 2017.
- [53] C. Schankula, E. Ham, J. Schultz, Y. Irfan, N. Thai, L. Dutton, P. Pasupathi, C. Sheth, T. Khan, S. Tejani, et al. Newyouthhack: Using design thinking to

- reimagine settlement services for new canadians. In *Innovations for Community Services: 20th International Conference, I4CS 2020, Bhubaneswar, India, January 12–14, 2020, Proceedings 20*, pages 41–62. Springer, 2020.
- [54] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, 1995.
- [55] D. Silver, M. Saunders, and E. Zarate. *What factors predict high school graduation in the Los Angeles Unified School District*, volume 14. California Dropout Research Project Santa Barbara, CA, 2008.
- [56] S. Smith, C. W. Schankula, L. Dutton, and C. K. Anand. A software engineering capstone course facilitated by github templates, 2024.
- [57] A. Sullivan and M. U. Bers. Computer science education in early childhood: The case of scratchjr. *Journal of Information Technology Education. Innovations in Practice*, 18:113, 2019.
- [58] M. F. Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. Yale University, 2000.
- [59] K. Trivodaliev, B. R. Stojkoska, M. Mihova, M. Jovanov, and S. Kalajdziski. Teaching computer programming: The macedonian case study of functional programming. In *2017 IEEE Global Engineering Education Conference (EDUCON)*, pages 1282–1289. IEEE, 2017.
- [60] L. S. Vailshery. Most widely utilized programming languages among developers worldwide 2023. <https://www.statista.com/statistics/793628/>

- worldwide-developer-survey-most-used-languages/, 2024. Accessed: 2024-05-14.
- [61] E. Wallingford. Functional programming patterns and their role in instruction. In *Proceedings of the international conference on functional programming*, pages 151–163, 2002.
- [62] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252, 2000.
- [63] G. Washburn and S. Weirich. Good advice for type-directed programming aspect-oriented programming and extensible generic functions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 33–44, 2006.
- [64] R. Wicentowski and T. Newhall. Using image processing projects to teach cs1 topics. *ACM SIGCSE Bulletin*, 37(1):287–291, 2005.
- [65] R. Wolfe. New possibilities in the introductory graphics course for computer science majors. *ACM SIGGRAPH Computer Graphics*, 33(2):35–39, 1999.
- [66] U. Wolz, H. H. Leitner, D. J. Malan, and J. Maloney. Starting with scratch in cs 1. In *Proceedings of the 40th ACM technical symposium on Computer science education*, pages 2–3, 2009.
- [67] P. Woodworth and W. Dann. Integrating console and event-driven models in cs1. *ACM SIGCSE Bulletin*, 31(1):132–135, 1999.



- [68] D. Yang, Z. Yang, and M. U. Bers. The efficacy of a computer science curriculum for early childhood: evidence from a randomized controlled trial in k-2 classrooms. *Computer Science Education*, pages 1–21, 2023.
- [69] J. Zhang, A. Verma, C. Sheth, C. W. Schankula, S. Koehl, A. Kelly, Y. Irfan, and C. K. Anand. Graphics programming in elm develops math knowledge & social cohesion. In *CASCON*, pages 157–167, 2018.