DO WIDGET LIBRARIES NEED MUTABLE DATA?

DO WIDGET LIBRARIES NEED MUTABLE DATA?

By AKSHAY KUMAR ARUMUGASAMY,

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

MASTER OF SCIENCE - COMPUTER SCIENCE (2023)

Hamilton, Ontario, Canada (Dept of Computing and Software)

TITLE:	Do widget libraries need mutable data?
AUTHOR:	Akshay Kumar Arumugasamy
SUPERVISOR:	Dr. Christopher K Anand
NUMBER OF PAGES:	xiv, 154

Abstract

This thesis examines trends in the academic and professional literature around immutable data and its relationship with declarative User Interfaces (UIs). Immutable data types are preferred by academic authors due to their increased safety, and commercial languages are increasing their support for them over time. More recently, declarative UIs are an exploding topic in industry, and these are related, although not as closely as one would expect. Declarative programming tries to focus on high-level requirements, not low-level details. It is easier to do this if functions have no side effects, and immutable data is a guaranteed way of achieving this. To highlight this property, the declarative UI framework Flutter advertises "stateless widgets", but their existence puts in highlights the lack of this property in most widgets. Consequently, we ask whether it is feasible to build a Graphical User Interface (GUI) toolkit using purely immutable data structures. To accomplish this objective, a purely immutable GUI toolkit is sketched and partially developed using Elm, a purely functional language in which all data structures are immutable. To understand the requirements of a GUI toolkit, we categorize and put in historical context, different design paradigms for UIs and relate them to core software-design principles. Leading toolkits allow developers to visualize and manage multiple views of their interfaces,

including the view hierarchy, layout, interface to business logic and focus management. By creating a concrete example, the research aims to provide insight into the limitations of utilizing purely immutable data within a GUI framework and suggests future work to mitigate these.

To my esteemed supervisor, Dr. Christopher Anand, my gratitude is beyond bounds to him. I extend my deepest gratitude to my cherished family members, baby Yaash, Vijay, Padma, Appa, and Amma, your presence in my life has been a constant source of strength and inspiration. Thank you for standing by me through the challenges, celebrating my successes, and reminding me of the importance of perseverance. Grateful to my uncle Dr. Thangamani Seenivasan for all his support throughout my journey. Finally, I express my heartfelt gratitude to the divine for gracing me with boundless blessings, guiding me through the ups and downs, and instilling the determination to pursue knowledge within me.

Acknowledgements

I would like to express my sincere gratitude to the summer students of Dr. Christopher Anand's research group particularly Haley Johnson, Zonna Mir, Christopher Schankula, Sheida Emdadi and Nasim Khoonkari for their invaluable assistance in reviewing this paper. I am deeply thankful to Dr. Christopher Anand for his unwavering patience, guidance, and confidence in my work throughout the past two years. Additionally, I extend my appreciation to the dedicated members of McMaster Start Coding and STaBL Foundation, who have tirelessly endeavored to enhance student exposure and confidence in Computer Science topics over the course of several years.

Furthermore, I wish to extend my heartfelt thanks to the Department of Computing and Software for providing me with this exceptional opportunity, their generous financial support and great lectures despite remote learning. Their contribution has been instrumental in the successful completion of this thesis.

Table of Contents

List of Figures

List of Tables

Abbreviations

ADT	Algebraic Data Types
CSS	Cascading Style Sheets
DFD	Data Flow Diagram
DOM	Document Object Model
Fran	Functional Reactive Animation
FRP	Functional Reactive programming
GOF	Gang of Four
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
MVC	Model–view–controller
MVP	Model–view–presenter
MVU	Model–view–update
MVVM	Model–view–viewmodel
OO	Object-Oriented

SVG	Scalable Vector Graphics
SoC	Separation of Concerns
SQL	Structured Query Language
TEA	The Elm Architecture
UI	User Interface
UX	User Experience
WYSIWYG	What you see is What you get
WWDC	Worldwide Developer Conference
XML	Extensible Markup Language

Chapter 1

Introduction

Software design principles have evolved to guide developers toward software that is more reliable and easier to adapt to new situations. The main principle, separation of concerns (SoC), leads to design patterns and frameworks. In Chapters 2 and 3, we explain these principles, focusing on SoC, and explore how design principles relate to the design patterns that almost all frameworks use. Each of the GUI architectures aims to separate the business logic from the mechanics of the user interface, also called the presentation logic. The most obvious realization of SoC is in the separation of software into modules, but this is not the only way to view the separation of concerns. We can also understand software in terms of its data flow and control flow, and User Interface (UI) software in terms of the view hierarchy. After introducing these concepts, we explain how the SoC principle is relevant to each of these views. These views are helpful because they contain concrete information which can be extracted automatically by development tools, making it easier for developers to apply SoC principles, than with other design patterns.

Another set of principles about which designers must make independent judgments

are Norman's Principles of interaction, which we define and relate to GUI frameworks in Section 2.6.

The top risk with mutable data structures is that they allow changes from one component to affect a data structure that another component is relying on to remain unchanged. This often leads to unexpected behaviours (bugs, crashes, and security vulnerabilities). To solve this vulnerability, we introduce immutable data and how it can play a role in improving the design of our software. In section 2.5, we explain the concept of declarative programming. We unpack the contrast between declarative and imperative approaches.

Chapter 4 demonstrates that a declarative style of programming is possible even in languages that are not fully declarative. This is especially important in the case of UI programming, which commonly uses libraries designed to support declarative UI written in a general-purpose programming language. We share the trends and evidence from major tool vendors showing strong movement towards declarative UI as their preferred graphical user interface development paradigm.

In Chapter 5, we summarise the evolution of GUI toolkits from the 1970s until today and some common challenges and solutions for immutability adoption in programming languages and frameworks. We explore the building blocks of the user interface, often called widgets, and explain how they structure toolkits. It is surprising that the number of toolkits is increasing and not converging to a perceived best practice. This can be explained by the introduction of web programming, in which the paradigms evolved from Smalltalk's Model-View-Controller were shoehorned into an environment (HyperText Markup) which was not designed for interactive programming. We are hopeful that a better GUI programming paradigm will evolve out of

this conflict.

It is in this context that we decided to attempt to create a purely declarative UI framework. In Chapter 6, we explain that the easiest way to show that this framework is feasible is to implement it in a language only supporting immutable data types, namely Elm. Although we have not implemented all of the widgets required in a useful GUI Toolkit, we have implemented enough to know that it is feasible, and to identify the challenges from the point of view of the toolkit developer and the toolkit user.

We can summarize the goals of this thesis in the following research questions:

RQ1 Is it practical to build a GUI Toolkit using purely immutable data (internally)?

RQ2 What are the advantages of Declarative UIs, from both academic and professional points of view?

RQ3 Are Declarative UIs the future?

Chapter 2

Design Background

Have you ever been in a situation where, while working on a project, you have suddenly thought of adding a new feature to your application, but you had the following questions:

1) How simple is it to modify the existing code?

2) How easily can it be done?

3) How much of the previous code structure or architecture can you utilize before disrupting existing functionality that other components of your system are using?

The best answer to these questions is the *Separation of Concerns* (SoC) principle. Separating the code into blocks that each govern a specific behaviour of the application limits the amount of code that needs to be modified when adding a new feature. This means that only the code that directly relates to the new functionality will need to be changed, overall resulting in a smaller number of code modifications. Dividing the responsibilities of the code helps in preventing any disturbance in unrelated features

by eliminating the need to make changes in the code that these features might use. It is less probable for code to malfunction if it does not require any unnecessary modifications. If the behaviours that you want to focus on are isolated and distinct from the other parts of the application, you will have a higher chance of being able to replace them with a new version without having to comprehend or modify the rest of the program. Moreover, it will be simpler to identify the code that needs to be modified.

But what is *separation of concerns*?

Separation of concerns is a guiding principle in software design; the idea behind this principle is to decompose a system into distinct, loosely-coupled parts, with each part responsible for a specific aspect of functionality. The goal is to create a modular and maintainable architecture by isolating different concerns and minimizing dependencies between them.

This modularity allows independent development, easier testing, and the ability to change one concern without accidentally affecting others. It enables the creation of flexible, extensible, and comprehensible software systems. The concept of separation of concerns played a significant role in the development of design patterns by the *Gang of Four* (GOF).Gamma et al. [65] in their influential book *"Design Patterns: Elements of Reusable Object-Oriented Software,"*emphasized the importance of modular design and separating concerns as a foundation for creating flexible and reusable software systems. The GOF recognized that software systems often consist of multiple concerns, such as handling user interfaces, managing data, or enforcing business rules. They identified recurring design problems and proposed ***design patterns*** as solutions to these problems.

2.1 Design Patterns

What is a design pattern, and how is it different from an architectural pattern or a framework?

Buschmann et al. [22] says an *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. The Model-view-controller discussed in the upcoming chapter would be an example.

Gamma et al. [65] in the book says a *design pattern* provides a scheme for refining the subsystems of a software system or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context. It is smaller in scale than architectural patterns, independent of a particular programming language but dependent on a programming paradigm. According to Buschmann et al. [22] the application of a design pattern has no effect on the fundamental structure of a software system but may have a strong influence on the architecture of a subsystem meaning it provides a description or template for solving the problem within a specific context, rather than being a ready-to-implement design that directly translates into source or machine code [172].

Buschmann et al. [22], Yacoub and Ammar [177] say a *framework* is a partially complete software system that is intended to be instantiated. It defines the architecture for a family of systems and provides the basic building blocks to create them. It includes a collection of libraries, tools, and conventions that help developers build software systems more efficiently.

Popular authors like Stokke et al. [158], Fowler [60], Fayad and Schmidt [53] say, a library is a collection of pre-written code that provides specific functionalities, and when using a library, the application developer writes the main body of the application and incorporates elements from the library as needed. On the other hand, a framework is a more comprehensive structure that provides the main body of an application, including predefined rules and conventions. In the case of a framework, the application developer focuses on implementing specific components or "hot spots" while the framework takes care of the overall control and flow of the application, often utilizing the principle of "inversion of control" to manage dependencies and provide extensibility. Frameworks emphasize the separation of concerns, where the data layer (referred to as the "model") and the presentation layer (known as the "view") are kept distinct. Frameworks define the interaction between these layers, ensuring a clear separation and organization of responsibilities.

Now, it is worth noting that there can be some overlap between design patterns and architectural patterns, as design patterns can be used within an architectural pattern to address more specific design problems within the context of a given architecture. They provide reusable solutions to common design issues at a lower level, often focusing on individual classes, objects, or interactions within a component. Frameworks, on the other hand, are typically larger in scope and provide a more comprehensive set of tools and guidelines to develop applications based on specific architectural patterns.

This image 2.1 illustrates the high-level concepts in software architecture:

Separation of concerns is closely tied to how data is handled and separated within a software system.

Figure 2.1: Design pattern in action

Data is an essential aspect that needs to be appropriately handled and separated within the system. Proper encapsulation, modularity, abstraction, and data flow management contribute to creating more modular, maintainable, and scalable systems.

Encapsulation refers to bundling related data and functionality together. By encapsulating data within the appropriate components, data is protected and only accessible to the relevant concerns. This helps maintain data integrity and ensures that data is not mishandled or accessed inappropriately.

Modularity refers to the proper separation of data, which allows for modular design. Each component should have access to the data it needs to perform its specific responsibilities, but it should not have access to data that is not relevant to its concerns. This modular approach enables easier development, testing, and maintenance of individual components.

Abstraction refers to creating abstractions and interfaces to define contracts between components. These abstractions often include data structures or data models that represent the shared understanding of the data between different concerns. By defining clear data abstractions, the concerns can interact with the data consistently and reliably.

Data flow plays a key role here. First, let's try to understand the 2W's of data flow.

2.2 Data flow and its significance

What is a data flow and why is important to understand it? A data flow is a flow or movement of information for a process or a system. It shows how data enters and leaves the system, what data is updated, and where data is stored. Some design patterns and architectural styles, like the "Pipes and Filters" pattern or the "Data Flow architecture", specifically address data flow management as a central aspect of their design as stated by Kumar [99].

Data flow can be visualized using dataflow diagrams. Using standardized symbols and notations, *data flow diagrams* (DFDs) illustrate the data movement involved in the operations of a business. Data flow diagrams are important in the design phase of software development (i.e. even before the software is written) because it helps the designer design the software and the developer understand the flow early in the design and represent it explicitly. It will also be useful for finding and clearing the faults of software in the early developing period. Properly separating and controlling the flow of data, ensures that each concern receives the data it needs to perform its tasks without unnecessary dependencies or coupling. This increases modularity, reduces complexity, and makes the system more maintainable, as claimed by DeMarco [42].

2.3 Separation of user interface concerns

Separation of user interface concerns specifically focuses on dividing the user interface-related functionality from other concerns in the system. It involves separating the presentation layer (UI) from the business logic and data processing layers. While separation of concerns is a more general principle applicable to various aspects of

software design, separation of user interface concerns is a specific application of that principle focusing on the separation of UI-related functionality from the rest of the system. In the field of software engineering, the importance of separating user interface concerns from purely functional aspects is widely acknowledged as stated by Browne et al. [20] in their book "Methods for Building Adaptive Systems" and also stated,

> "It is widely appreciated that the separation of user interface concerns from purely functional concerns is good software engineering practice" - Browne et al. [20].

We will delve into some of the most widely recognized graphical user interface (GUI) architectures that have been in use by professionals for a considerable period in Chapter 3. These architectures are continually evolving and being restructured as necessary to meet the demands of modern software development. The main aim of these architectural patterns is to help with the separation of concerns; the code should be divided into modules that allow developers to easily understand the parts of the system where a change is required. While separating it into modules, let's consider how each of the architectural patterns achieves it based on its *data flow, control flow, view hierarchy, and module decomposition.* We have discussed data flow in section 2.2.

Control flow refers to the order in which instructions are executed in a program. It determines the sequence in which statements are executed and how control is transferred between different parts of the program, as stated by Watt [164].

View hierarchy refers to the organization of views in a software system. Views are the user interface components that allow users to interact with the system. The view hierarchy determines how views are organized and how they interact with each other.

Module decomposition is the process of breaking down a software system into smaller, more manageable modules. Each module should have a well-defined interface and be responsible for a specific set of tasks. This approach makes it easier to develop, test, and maintain software systems.

2.4 Understanding Mutability and Immutability

Mutate means to 'change'. So mutable means 'able to change'. According to Wikipedia [171] Mutation allows for *side effects*. An operation, function, or expression is said to have a side effect if it modifies some state variable value(s) outside its local environment, which is to say if it has any observable effect other than its primary effect of returning a value to the invoker of the operation. A system is described as stateful if it is designed to remember preceding events or user interactions, the remembered information is called the state of the system. Side effects play an important role in the design and analysis of programming languages. The degree to which side effects are used depends on the programming paradigm. For example, imperative programming is commonly used to produce side effects and to update a system's state. By contrast, declarative programming is commonly used to report on the state of the system, without side effects and is discussed more in section 2.5.

Based on the concepts and solutions suggested by these authors Fleury [54], Hughes [89], Zub [180] we will take an example in order to explain the concept, let's break it down from a functional perspective. Imagine a user interface as a function that takes an initial state and produces a new state. When writing code that mutates the state directly, there's a problem that arises. If you modify the state in the middle of implementing the user interface, other parts of the interface might not

be aware of this change and could be making false assumptions about the state. This discrepancy in opinions about the state can lead to issues in the user interface. For instance, you might see text appearing outside its designated area, multiple elements moving together but not synchronously, or graphical glitches that appear briefly and then disappear.

Let's take a simple example to illustrate this issue. Consider a button in a user interface meant for deleting an object. If the actual deletion of the object happens immediately when the button is clicked, any code that still references that object afterwards will either break or potentially crash the program. In essence, separating user interface concerns from purely functional concerns is important to avoid these types of problems and ensure a more reliable and consistent user experience. In some functional programming languages, the lack of side effects is further strengthened by the fact that there are no variables or assignments. Since no variables exist, there is no possibility of side effects. The concept of 'purity' is also heavily explored in a functional programming language. A pure function only accepts a value and returns a value. Pure functions do not rely on any global states. As a direct consequence of functions being side-effect free and pure, a repeated call to a function with the same arguments returns the same value and this is known as *referential transparency* [82]. A referentially transparent function is one which only depends on its input [144]. This is why it is hard to achieve in OO programming, because objects have a state, as rightly pointed out by Kunasaikaran and Iqbal [100], Quine [141]. With mutations, the reliability of the system is questionable because it is harder to verify soundness. This would not be the case if we use *immutable data*.

Immutable data

In the paper "A foundation for user interface construction" by Myers [123], Gansner emphasizes the significance of adopting a side-effect-free programming style, particularly in concurrent systems where issues of interference may arise. Gansner states,

> "Side-effect-free programming style should be the norm. This is especially important in concurrent systems, where issues of interference arise. Furthermore, the use of 'pure' functions and immutable data greatly increases the clarity and reliability of programs." -Myers [123]

Imagine you have a permanent marker in your hand. Since the ink is permanent, when you use this marker to write something on a whiteboard, it cannot be erased or modified. Whatever you write remains unchanged, just as if it were carved in stone.

Now, let's relate this concept of immutable data to programming. In software development, immutable data refers to data that cannot be altered after it is created, much like the markings made with a permanent marker. Once the data is initialized, its value remains fixed and cannot be changed.

Just as the ink from a permanent marker cannot be erased, immutable data ensures that the state of an object remains consistent throughout its lifetime. This predictability eliminates unexpected changes or side effects that could occur if the data were mutable.

The use of immutable data offers several advantages. Firstly, it makes the behaviour of the code easier to reason about. Since the data cannot be modified, you can confidently understand how it will be used and what effects it will have on the program. There are no surprises due to unexpected modifications.

Secondly, debugging becomes simpler with immutable data. If there is an issue, you can trace it back to a specific point in the code where the immutable data was created or used. Because the data remains unchanged, you can examine its values and properties with certainty, making it easier to identify and resolve any bugs or errors.

Hickey [86], the creator of Closure programming mentions "Immutable data also plays a vital role in concurrency and parallelism." Just as the markings made with a permanent marker are resistant to smudging, immutable data is inherently thread-safe. It can be accessed and read by multiple threads simultaneously without the risk of race conditions or data corruption.

Furthermore, immutable data aligns well with functional programming principles. Kaya [93] stated "Functional programming emphasizes the use of pure functions that do not have side effects." Immutable data facilitates the creation of pure functions because they only rely on their inputs and produce deterministic outputs. This makes the code easier to understand, test, and maintain.

Lastly, just as markings made with a permanent marker cannot be erased, immutable data retains its value over time. This can be useful in scenarios such as historical tracking or maintaining a log of changes. Each change creates a new immutable instance, allowing you to track the history of data transformations and easily access previous states.

In summary, the permanent marker analogy helps to illustrate the concept of immutable data. Immutable data, like permanent ink, ensures that the state remains unchanged once it is created. This predictability leads to easier reasoning, simpler debugging, safe concurrent processing, alignment with functional programming, and

the ability to track data history. Of course, my analogy skims only the thinnest surface of the deep waters of the concept of immutability. We will discuss more about this in later chapters.

2.5 Declarative Approach Vs Imperative Approach: Unpacking the Contrast

Imagine you are going out to eat at a restaurant. The menu is like a recipe book, listing all the available dishes with their ingredients and instructions on how to prepare them. In this case, the menu is akin to an *imperative approach*. With an imperative approach (like following a recipe book), you must go through each step and perform them in a specific order to achieve the desired outcome. It's a detailed set of instructions that guide you through the cooking process.

Now, let's switch to a *declarative approach*. Instead of ordering from a menu or following a recipe book, imagine you have the opportunity to describe your ideal meal to the chef. You can specify your preferences, such as the type of cuisine, the ingredients you like, and any dietary restrictions. Based on your description, the chef will use their expertise to create a customized dish that suits your tastes and requirements. In this scenario, the declarative approach is like describing your desired outcome (the meal) without explicitly stating the steps to achieve it. You focus on communicating your intentions and letting the chef (or the underlying system) figure out the best way to deliver them. Similarly, in software development or problem-solving, a declarative approach involves specifying the desired outcome or behaviour without getting into the nitty-gritty details of how to achieve it. You define the goal

or state you want to achieve, and the system determines the most suitable approach to accomplish it, leveraging its internal knowledge and capabilities. By embracing a declarative approach, developers can focus more on expressing the desired outcomes, which can lead to increased flexibility, modularity, and automation in building complex systems. It allows for a higher level of abstraction and empowers the system to make decisions based on the defined goals as mentioned by Cook [30], similar to how a chef creates a dish based on your meal description.

A **declarative approach** is a programming paradigm or methodology that focuses on expressing the desired outcome or goal of a program, rather than explicitly specifying the steps or procedures to achieve it. It emphasizes the "what" rather than the "how" of the program, as described by Alpuente et al. [2]. **Declarative programming languages** are languages specifically designed to facilitate this approach by providing constructs and syntax that allow programmers to express programs declaratively.

Skoczylas [153] explains **Declarative programming paradigms** include the following :

- **Functional programming** focuses on writing programs by composing pure functions, which do not have side effects and produce the same output for the same input. It emphasizes immutability and the use of higher-order functions to manipulate data. Functional programs contain no assignment statements. This means that variables, once given a value, never change. More generally, functional programs contain no side effects at all. A function call can have no effect other than to compute its result. This eliminates a major source of bugs and also makes the order of execution irrelevant since no side effect can change

an expression's value, it can be evaluated at any time, as said by Hughes [89].

- **Logic programming** is based on formal logic and allows programs to be written in terms of rules and logical relationships. It employs inference and logical deduction to derive results from given facts and rules.
- **Database query languages**, such as SQL, are declarative in nature. They allow users to express queries to retrieve or manipulate data from a database without specifying the exact steps to achieve the desired results.
- **Markup languages** like HTML and XML are declarative in nature. They describe the structure and presentation of documents or data, rather than providing explicit instructions for rendering or processing.

2.6 Don Norman's Principles and Their Relevance to Immutability and Declarative Approaches

> "Good design is actually a lot harder to notice than poor design, in part because good designs fit our needs so well that the design is invisible," — Donald A. Norman, The Design of Everyday Things[125]

Don Norman's principles [125] of interaction design offer valuable insights into creating human-centered software. When exploring the connections between these principles, immutability, and the declarative approach, we find a rich interplay that enhances usability and system comprehension.

1. **Visibility**: Visibility refers to the clarity and transparency of a system's state and behavior. *Immutability*, in the context of object-oriented programming,

contributes to visibility by ensuring that the state of an object remains constant and hence the view only needs to be drawn once. On the programming level, immutability allows developers to easily understand and reason about the system's behaviour, as no unexpected changes are occurring. It is hard for developers to communicate state with users if they do not fully understand it. Similarly, the *declarative approach* fosters visibility by explicitly defining the desired outcomes, making it easier to communicate the intentions and goals of the system.

2. **Feedback**: The book says "Feedback is essential for users to understand the consequences of their actions." Norman [125]. Mutable data structures allow for changes in data structures to occur in unexpected places in the code. When unplanned pathways are used to update underlying data, mechanisms to reflect that change in the interface can be bypassed, resulting in a failure to provide feedback for user actions. Immutable data structures must be reconstructed when they change, and it is much easier to ensure that this only happens in ways that lead to recreated visible widgets.

3. **Constraints**: Constraints guide users toward appropriate actions and prevent unintended errors. *Immutability* acts as a constraint by limiting the ability to modify an object's state after it is created, making it harder to accidentally create situations in which user actions could destroy data or corrupt on-going computation.

4. **Mapping**: Mapping refers to the relationship between controls and their actions or effects. Some declarative approaches create mappings in the source code. For instance, Fudgets (to be discussed in 5.4) uses infix operators and data flow to represent the flow of signals and data within GUI components demonstrating how the visual representation and declarative nature of fudgets enhance the readability and

comprehension of the code like it follows a pipeline structure.

5. **Consistency**: Consistency ensures that elements and interactions have a uniform appearance and behaviour. Declarative approaches enhance consistency by specifying standardized behaviours and outcomes based on intent. By defining functionality at a higher level, the programming library translates it into actions consistently across the system. This approach reduces the reliance on individual programmers to enforce consistency manually, as the declarative nature of defining behaviours ensures uniformity. For instance, if all widgets are defined in terms of their functionality rather than appearance, consistency is automatically achieved. This is similar to the original idea of HTML, where tags defined the structure of a document, but CSS was introduced to allow flexible layout changes. By focusing on functionality rather than appearance, declarative approaches encourage consistency without relying solely on the programmer to enforce it.

6. **Signifiers**: Signifiers, also known as affordances, refer to the use of visual elements associated by users with their function. This is not directly related to immutability.

By incorporating these principles into system design, immutability, and the declarative approach contribute to user-friendly experiences, better system comprehension, and more consistent and intuitive interactions. Understanding the relationship between these principles and their integration can inform the creation of systems that align with human cognition and behaviour, ultimately leading to enhanced usability and satisfaction.

Chapter 3

Graphical User Interface Architectures

> "Architecture is the set of design decisions that must be made early in a project." - Ralph Johnson as quoted by Fowler [59] in his article.
>
> "The highest-level concept of a system in its environment. The architecture of a software system (at a given point in time) is its organization or structure of significant components interacting through interfaces, those components being composed of successively smaller components and interfaces."- IEEE Std 1471-2000[154], Fowler [59]

This section will consist of popular GUI architectural patterns discussed by one of the well known authors Fowler [61] in his article.

3.1 Forms and Control

In a GUI application, the form is responsible for the layout and behaviour of the controls on the screen. The controls display data, which can come from a database and exist in three copies:

- the record state in the database,
- the session state in memory, and
- the screen state is the data they see on the screen.

Data binding helps to keep the session state and screen state synchronized, but it cannot manage all behaviour. Events allow the form to be notified of changes in the controls and to carry out specific behaviour through routines. Data binding and events together ensure the correct display and behaviour of the controls on the screen.

3.2 Model–View–Controller (MVC)

Model–view–controller (MVC) as shown in 3.1 is a design pattern for graphical user interfaces that originated in the 1970s. Trygve Reenskaug created MVC while working on Smalltalk-79 as a visiting scientist at the Xerox Palo Alto Research Center (PARC) [169]. It was one of the first attempts to build UI systems on a large scale. Potel [138] states "The key idea behind MVC is textitSeparated Presentation, which divides the system into two parts: the domain model and the presentation."
The *domain model* consists of objects that model the real world, while the presentation consists of the GUI elements on the screen. The *presentation* part of MVC is made up of the view and controller. The *controller* takes the user's input and decides

what to do with it, while the view represents the appearance of the GUI element on the screen. The view and controller are generic components that can be reused.

- Model: The domain model is a representation of the real-world objects and rules that govern a particular problem domain. It contains all the interesting data and logic of an application, and it is completely ignorant of the user interface (UI).

- View: The view is responsible for presenting the model's data to the user in a way that makes sense for their needs. It should be as passive as possible, meaning it has no knowledge of the model or any other part of the system. The data is provided by the controller through an interface.

- Controller: "The controller acts as an intermediary between the view and model." Kuzmenko [101]. It handles user input and updates both as necessary. It is also responsible for managing control flow within an application.

The key thing to remember is that the *data* which the software application is processing is known as the 'model' (a.k.a. Application State, Domain Model, or Business Logic) of the application. The 'view' represents this data to the user in a concrete manner and the user can interact with it. Here the controller determines the path a program takes and which parts of the code are executed based on certain conditions. Hence, the controller has access to both the model and the view; it queries the model to apply the business logic based on the action requested, receives the updates of the model, and updates the view with the current model. Once the view is updated, the user can see what changes have been made. The important thing to note here is that the developer is responsible for ensuring that every model update is reflected in

the view. According to Martin Fowler, a renowned software development expert, the concept of Model-View-Controller (MVC) can be subject to varied interpretations by different individuals, leading to divergent understandings of MVC. Fowler states,

> "Different people reading about MVC in different places take different ideas from it and describe these as 'MVC'. If this doesn't cause enough confusion you then get the effect of misunderstandings of MVC." [61]

Fowler says MVC is one of the most misunderstood patterns in the software world, understandably since it is not well-documented[62].

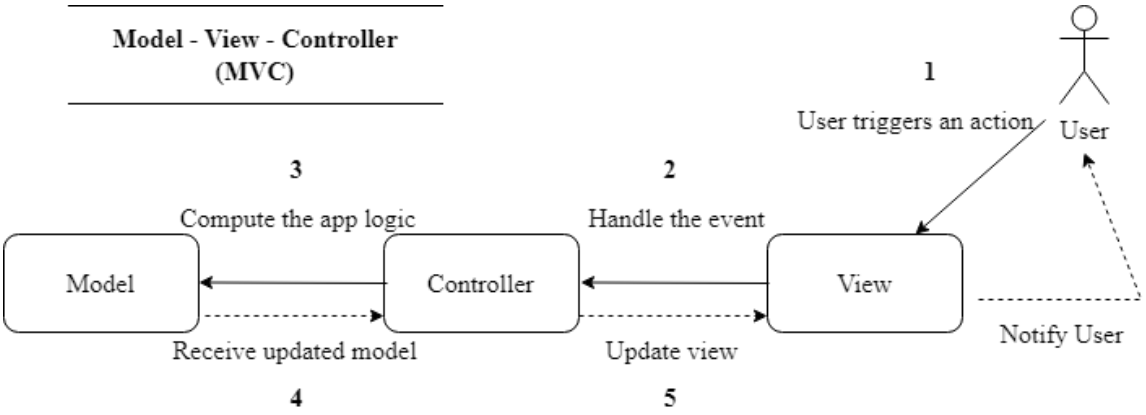

Figure 3.1: Model-view-controller (MVC)

In alignment with the aforementioned statement, my research on Model-View-Controller (MVC) resonates with the notion that diverse interpretations of MVC exist within the software development community. This observation is substantiated by the abundance of blog posts I encountered during my investigation, wherein various authors presented MVC from distinct application perspectives. For instance, discussions ranged from the implementation of desktop application-based MVC to web-based MVC inspired by Smalltalk 80 and Apple's MVC architectural pattern.

In the Appendix A, we will explore a practical illustration of the discussed concepts through an implementation example.

3.3 Model–View–Presenter (MVP)

MVP as shown in 3.2 is an architectural pattern that initially emerged at IBM and gained more prominence during the 1990s, particularly at Taligent. The pattern was later migrated by Taligent to Java and popularized in a paper by Taligent CTO Mike Potel. The MVP pattern aims to bring together the best of both Forms and Controller and MVC architectures.

According to Potel [138], the 'view' is a structure of widgets that correspond to the controls of the Forms and Controls model (3.1) and removes any view/controller separation. It does not contain any behaviour that describes how the widgets react to user interaction. The presenter then decides how to react to the event. The idea is to separate the presentation logic (i.e. how data is displayed to the user) from the business logic (i.e. how data is processed and stored).

In MVP, there are two main variations of separated presentation: **Passive View** and **Supervising Controller**. In *Passive View*, the view is as passive as possible, with all UI-related logic handled by the presenter. The view has no knowledge of the model or any other part of the system and simply exposes a set of properties that can be read and written by the presenter. In *Supervising Controller*, on the other hand, the presenter acts as a mediator between the view and model, handling user input and updating both as necessary. This means that in *supervising controller*, the presenter has some control over the view, and in *passive view*, the presenter has complete control over the view.

The presentation layer does not depend on the UI as it does in MVC. Instead, the presenter communicates with the model layer through interfaces and receives model events back through interfaces as well. Therefore, the presenter will implement model layer interfaces and will communicate with the model layer. As the presenter receives model layer notifications, the user will create view data and directly pass it to the view. The view is then rendered with the updated model on the screen.

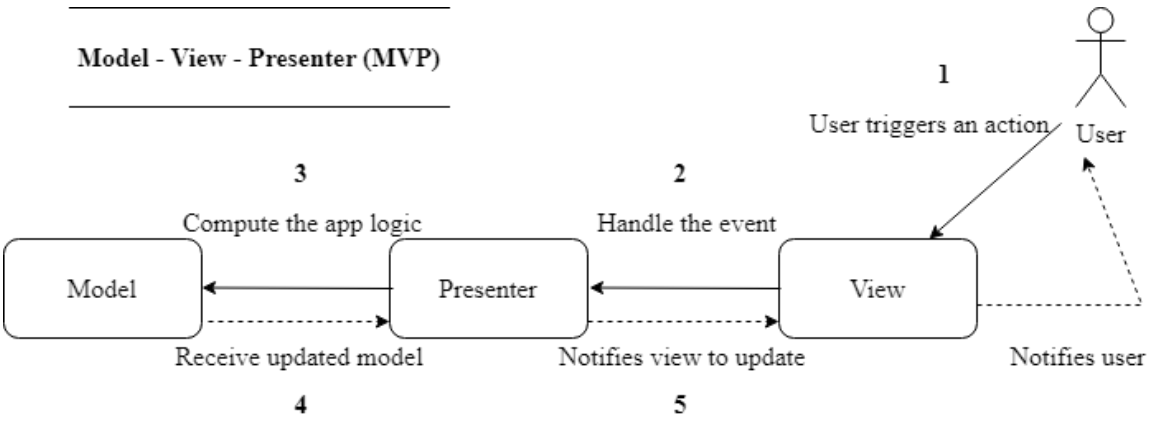

Figure 3.2: Model-view-presenter (MVP)

3.4 Model–View–ViewModel (MVVM)

MVVM as shown in 3.3 is a variation of Martin Fowler's [63] "presentation model design pattern". Microsoft architects Ken Cooper and Ted Peters invented it specifically to simplify event-driven programming of user interfaces. The pattern was incorporated into the Windows Presentation Foundation (WPF) [166].

The model is relatively similar to the one in both MVC and MVP, but here we have viewmodels that are passed to the view. All the logic is in the viewmodel and hence no controller/presenter is present.

- Model: The model represents the data and business logic of the application. It can include classes or structures that define the data objects, as well as methods for performing operations on that data, According to D. [39].

- View: The view is responsible for presenting the user interface to the user. The view observes the viewmodel for changes and updates the UI accordingly.

- Viewmodel: "The viewmodel acts as a mediator between the model and the view." Muliyashiya [120]. The viewmodel handles user interaction. It exposes the necessary data and methods required by the view to display and interact with the data. It typically exposes properties and commands that the view can bind to. In our case, the viewmodel would contain properties to hold the input numbers, a method to perform addition using the model, and a property to hold the result.

User events still come from the view and through a binding. User events are handled by the viewmodel, and the viewmodel passes the messages through the model. When the model changes, the viewmodel, through the observable properties, will notify the view.

As you can see here, the significant difference is that the viewmodel does not have a reference to the view directly. Earlier, the controller called the reference to the view, and normally the view also does not hold a reference to the viewmodel. There is something in-between these components allowing them to be connected, which is called a binder. A *binder* is a component or tool that connects two or more software components. The purpose of a binder is to enable communication and data exchange between different parts of an application or system, as stated by Gaudioso [66].

When MVVM was created by Microsoft, they were simply attempting to eliminate the boilerplate for connecting the presentable data to the view. At the time, they used something called XAML in a binder framework to keep the view up-to-date with the model without the developer having to write any code. MVVM was attempting to solve the same problem as MVC, but with less boilerplate code, and introduced an automated binder framework to show how the data within the viewmodel should be displayed, according to Gaudioso [66].

In MVC, for example, data binding can be used to connect a form field (view) with a corresponding property on an object (model). When the user enters data into the form field, that data is automatically updated in the corresponding property on the object[132]. Similarly, if the value of that property changes elsewhere in the application (such as through a database update), that change is automatically reflected in the form field.

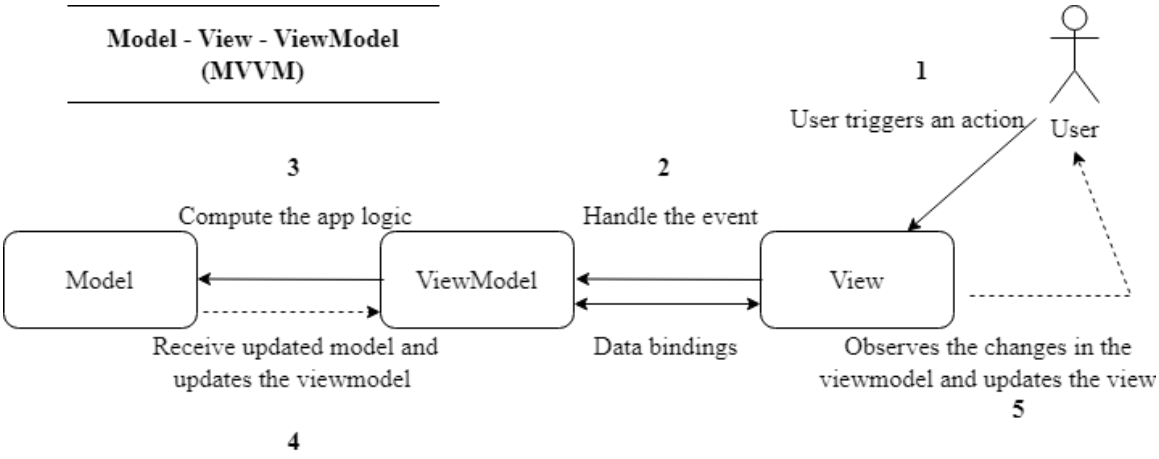

Figure 3.3: Model - view - viewmodel (MVVM)

3.5 Model–View–Update (MVU)

MVU as shown in Figure 3.4 is less well-known than other mentioned GUI architectures, and in fact, lacks a Wikipedia page. It was introduced in an update to the Elm language and is also referred to as The Elm Architecture (TEA) [34].

Elm as initially designed by Evan Czaplicki as part of his Harvard University thesis in 2012 used 'Functional Reactive Programming' (FRP) [35]. Elm was positioned as a gateway to functional programming for web developers—including self-taught developers—and functional reactive programming concepts were perceived as a barrier as per the paper Czaplicki and Chong [38]. Czaplick also introduced TEA as a simpler pattern for architecting web apps and Elm is no longer a FRP as decleared by Czaplick [36]:

- Model: The application state in Elm is represented by a data structure called the 'model'. It holds all the necessary information about the current state of the application. The model is *immutable* (further explained in 2.4), meaning it cannot be directly modified. Instead, when an update is needed, a new model is created that represents the updated state.

- Update function: To update the model, the user provides an 'update' function. This function takes the current model and an action (an instruction for the update a.k.a messages) as input and returns a new model as output. The update function is responsible for handling different actions and updating the model accordingly.

- View function: The view function in Elm is responsible for describing the user interface based on the current model. It takes the model as input and returns a

description of the UI. This description is essentially a pure function that maps the model to UI components and their properties.

- Document Object Model (DOM): The DOM is an interface that represents web documents as a structured collection of nodes and objects, corresponding to a HyperText Markup Language (HTML) document. It allows programming languages like JavaScript to interact with and modify the structure, style, and content of web pages dynamically as inferred from Mozilla.org contributors [117, 116, 115]

- Virtual DOM diffing: Virtual DOM is a programming concept used in web development frameworks that helps optimize the process of updating the user interface. Instead of updating the actual DOM every time a change is made to the user interface, the framework updates a virtual representation of the DOM and then compares it with the previous version to determine which changes need to be made to the actual DOM. This process is more efficient than updating the entire DOM every time a change is made, which can lead to slow and unresponsive user interfaces. In Elm, once the view function generates the UI description, Elm uses a technique called "virtual DOM diffing" to efficiently update the real DOM (the actual HTML elements rendered in the browser). The virtual DOM is a lightweight representation of the actual DOM, and by comparing the previous and current virtual DOMs, Elm identifies the minimal set of changes required to update the UI.

  According to the news released by the creator of Elm Czaplicki [37], as per the 2016 report their virtual DOM allowed the language to render HTML faster than the popular JavaScript frameworks *React*, *Ember*, and *Angular*.

Messages (`Msg`s) are the way an Elm program communicates with the world outside of Elm. Msgs come from events, like user interaction such as clicks on HTML elements, and IO events like HTTP requests and ports. The `update` is the central place to interpret all incoming messages as changes to the model.

Elm could have picked an architecture with many callback functions being passed to the runtime for it to call, but having a central place to deal with things that can change the model makes it easier to see how and why the model can change and this concept is discussed as one of the advantages of elm in this forum [91]. By following this flow, Elm achieves a controlled and predictable update process for the user interface. The functional and declarative nature of Elm helps to eliminate many common sources of *side effects* and provides a clear separation between state management and UI rendering. This approach leads to more maintainable, and reliable front-end code.

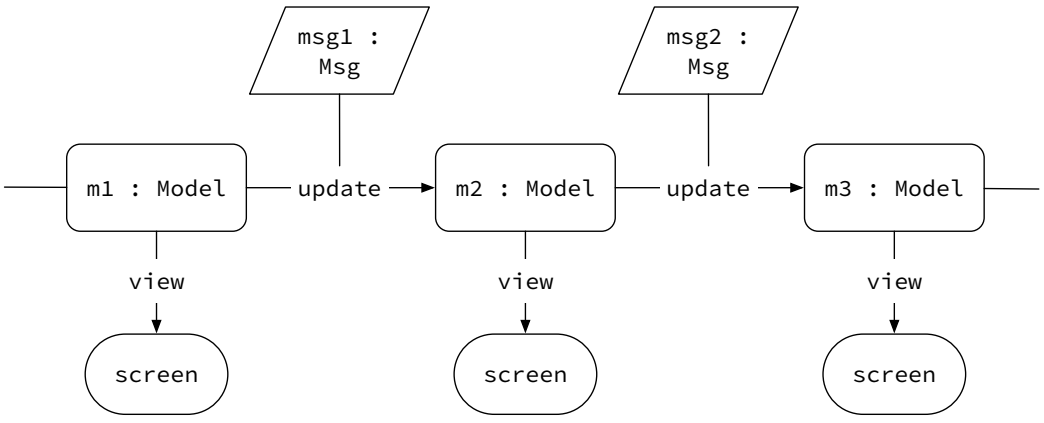

Figure 3.4: Model-view-update (MVU)

3.6 Analysis of MVC, MVP, MVVM, and MVU

Table 3.1: Analysis of MVC, MVP, MVVM, and MVU

Concepts	**MVC**	**MVP**	**MVVM**	**MVU**
Data Flow	**Bidirectional**. The user interacts with the View, which sends user input to the Controller. The Controller updates the Model. The Model notifies the View of any changes.	**Bidirectional**. The user interacts with the View, which sends user input to the Presenter. The Presenter updates the Model and notifies the View of any changes.	**Bidirectional**. User interactions update the viewmodel, which in turn updates the Model. Any changes in the Model are propagated to the View for display. With a Redux-like approach, we can make MVVM unidirectional [1].	**Unidirectional**. Update function takes the current Model state and user input to produce an updated Model state.

Continued on next page

[0]In MVVM, we would still require calling some method on the ViewModel when the user interacts with the View, but we can create an indirect link that does not directly interact with the ViewModel

Table 3.1 – Continued from previous page

Concepts	**MVC**	**MVP**	**MVVM**	**MVU**
Control Flow	Controller handles user input	Presenter handles user input	viewmodel handles user input	Update function handles user input
View Hierarchy	Views are passive, display-only	Views are passive, display-only	Views are passive, display-only	Views are passive, display-only
Module decomposition	Model, View, Controller are separate modules	Model, View, Presenter are separate modules	Model, View, viewmodel are separate modules	Model and Msg are types, View and Update are functions

We have analyzed the data flow, control flow, view hierarchy, and module decomposition in each of these patterns as discussed in Chapter 2. The paradigms above collectively underscore the significance placed on segregating business logic from presentation logic. Within the MVU pattern, the model is immutable, generating a new model whenever updates occur. This unidirectional flow of data provides a heightened level of predictability. By enforcing changes to propagate solely from the model to the view, the MVU pattern simplifies data flow, enhances comprehensibility, and

as described by Santos [148], Rambhia [142]

facilitates efficient data update management. As a result, this one-way flow engenders a more controlled and dependable system.

Chapter 4

Is The Future Declarative?

In declarative programming, programs focus on specifying *what* needs to be accomplished rather than explicitly defining *how* to achieve it. As a result, the program's behavior can be described in a more abstract and concise manner, as described by Ghezzi and Jazayeri [67].

Declarative programming is a programming paradigm where, at its core, it gives a set of declaration or declarative statements, each of which has a specific meaning in the problem space (all information that defines the problem and constrains the solution, the constraints being part of the problem). These can be understood independently or in isolation. A declarative programming language allows the expression of a program by breaking it down into multiple smaller statements, each representing a fact, opinion, or belief about the program's behavior. Instead of one large declaration, a declarative language enables the programmer to express the program's logic through a collection of individual statements, each contributing to the overall behavior of the program, according to Bmbarbour [18].

4.0.1 Declarative Approaches: Not Exclusive to Declarative Programming Languages

A language does not necessarily need to be exclusively declarative to enable declarative approaches. Many programming languages support a combination of declarative and imperative programming styles. For example, libraries written in *JavaScript*, *Python*, and *Ruby* allow programmers to write code in a declarative manner, even though the languages use mostly or entirely imperative programming constructs.

The key aspect is the programming style and methodology employed by the programmer, rather than the language itself. However, using a dedicated declarative programming language provides a focused and expressive environment for following declarative approaches, and enforces some aspects of declarative programming.

4.1 Declarative UI: Exploring the Trend and Future Potential

Imagine you are furnishing a room in your house. In a traditional, imperative approach, you would manually arrange and rearrange the furniture, constantly checking and adjusting the position of each item to achieve the desired layout. Now, let's consider a declarative approach using a declarative UI. Imagine you have a magic interior design tool that allows you to simply describe your required room functionality and desired appearance, and the tool automatically arranges the furniture accordingly.

In this analogy, the furniture represents the different user interface components like buttons, input fields, or images. The imperative approach would involve manually specifying the position, size, and behavior of each component, much like manually

moving the furniture in the room. However, with a declarative UI approach, you would describe the desired layout and appearance of the user interface using a high-level description, similar to how you would describe the desired room layout to the interior design tool. You might say things like, "I want a button at the top-right corner," or "I want a text input field in the center."

The declarative UI framework or library then takes this high-level description and automatically handles the rendering and positioning of the user interface components accordingly, similar to how the magic interior design tool automatically arranges the furniture based on your description. It simplifies the development process and allows you to focus more on the desired outcome rather than the detailed steps of how the UI should be constructed.

You can expand this concept to web app development. In a traditional imperative approach to web app development, developers would manually manipulate the Document Object Model (DOM) to create and update the user interface. They would write code that explicitly adds, removes, or modifies DOM elements to reflect the desired state of the UI. On the other hand, with a declarative UI approach like React, developers describe the UI structure and behavior declaratively, and the framework takes care of rendering and updating the DOM accordingly. Developers define components that represent different parts of the UI and describe how they should look and behave based on the application's state.

Section A.2, shows an implementation made using Android's XML layout and Jetpack Compose to demonstrate that when using a declarative approach, there are fewer lines of code. Also, it is easier to test, debug and maintain. It requires a considerable amount of effort to learn the syntax initially, but once learned it results

in simple and easy-to-understand code. Unlike Android XML, Jetpack Compose is written in Kotlin which can also be used to write the business logic. The UI elements can be reused and composed together, also within the Kotlin code. Thereby we have a single source of truth.

Similarly, Paul Hudson, in this video[88], states "In SwiftUI your data and the view hierarchy displaying your data are much more loosely coupled than in UIKit." and also, illustrates this comparison.

4.1.1 Industry Trends

Apple [10] at WorldWide Developer Conference(WWDC) said "SwiftUI and Swift are the future of development on Apple's platforms"[5]. Also, survey conducted by Siemens [152], shows a trend in organizations toward adopting low-code technologies which goes well with the announcements made at Apple's WWDC and echoes with the blog post by Evans [52]. Other organizations have also invested in declarative approaches, starting with Facebook's *React Native*, Google's native Android development using *Jetpack Compose*, Google's cross-platform development kit, *Flutter*, and Microsoft's *Fluent UI*.

In 2008, StackOverflow started collecting and analyzing post content [157], from which we can see a significant **increase** in interest in a declarative style of programming. We used their data to compare a more declarative framework with a less declarative framework, each targeting the same platform. In Figure 4.1 we see that Flutter has overtaken React Native. In Figure 4.2 we see that Jetpack Compose has recently rocketed ahead of the Android XML layout. In Figure 4.3, we see the most dramatic jump in interest in SwiftUI compared to the conventional UIKit. Google

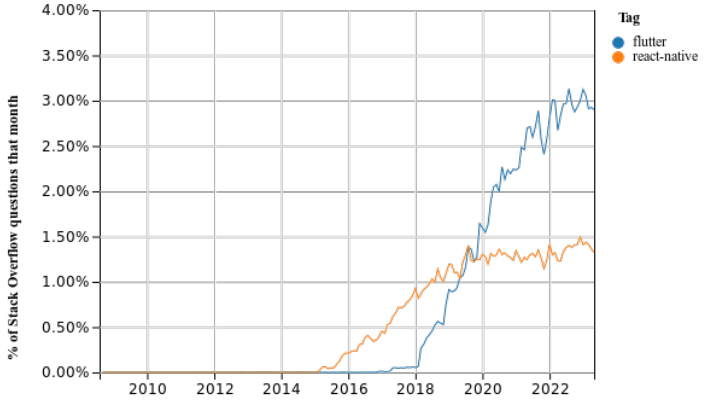

Figure 4.1: Trend highlights of Flutter over React Native

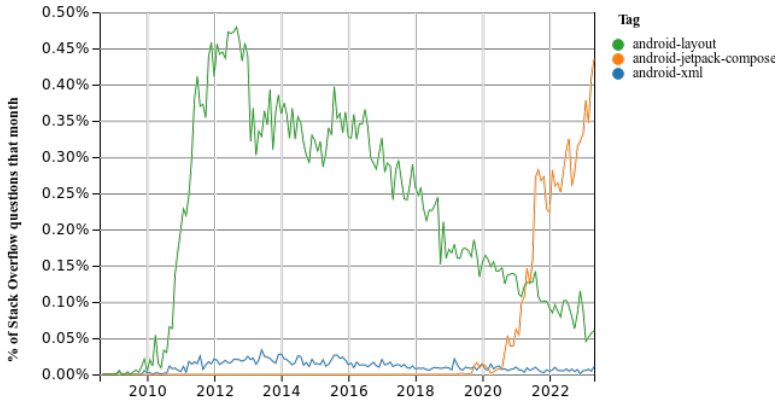

Figure 4.2: Trend highlights of AndroidXML Layout over Jetpack Compose

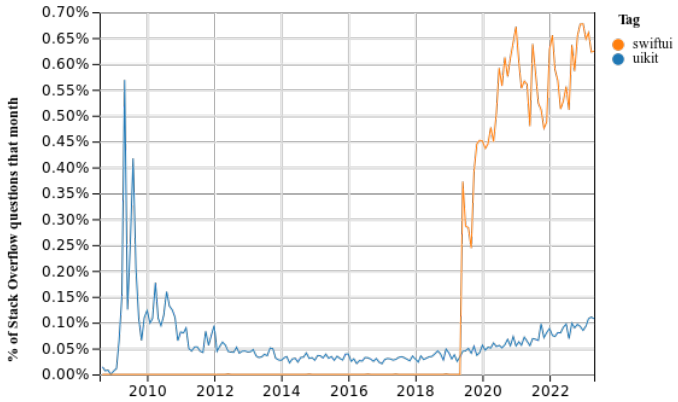

Figure 4.3: Trend highlights of UIKit over SwiftUI

Trends [76] echoes these results.

4.1.2 Exploring the Advantages of Declarative UI Programming with Prominent Platforms

"The declarative style of UI programming has many benefits. Remarkably, there is only one code location for any widget" as stated in Google [74] you simply describe what the UI should look like for any given state once, and nothing more the concept to achieve this is explained in there documentation "Start thinking declaratively". In the context of user interface design in Flutter, the equation "$UI = f(State)$"

Figure 4.4: Reflecting App State in the User Interface [74]

means that the user interface (UI) is a function of the current state of the system or application. In other words, the appearance and behavior of the UI are determined by the state of the underlying system [74].

To explain this concept further, consider a web application with different screens or pages. The UI of each screen can vary depending on the current state of the application. For example, if a user is viewing their profile page, the UI may display their profile picture, name, and relevant information. However, if the user switches to a different page, such as a settings page, the UI will change to reflect the new state and display different options and controls related to settings.

The equation *UI = f(State)* where `f` is a build method, says that the UI is not static but dynamic. It implies that the UI elements, layout, and content are computed based on the application state, allowing the developer to focus on the business logic, and let the framework update the interface.

This concept is often utilized in user interface frameworks and design patterns to make it easier to create responsive and context-aware interfaces that enhance the user experience. By dynamically updating the UI based on the system's state, designers and developers can provide users with interfaces that are intuitive, efficient, and closely aligned with their current needs and goals.

Apple [9] documentation states, "Declare the content and layout for any state of your view. SwiftUI knows when that state changes, and updates your view's rendering to match."

"**FluentUI** aims to support both models (imperative and declarative), but strongly recommends describing user interfaces using the declarative approach."—Microsoft Fluent Wiki [113]

"**Android Jetpack Compose**, over the last several years, the entire industry has started shifting to a declarative UI model, which greatly simplifies the engineering associated with building and updating user interfaces."—Google [68]

4.2 How Does Declarative Programming Relate to Immutability?

By explicitly expressing the relationship between the application state and UI functionally, declarative UI expresses concisely the idea that pure functional programming

should be used to connect business logic and GUI widgets. In declarative programming, there is also an emphasis on immutability, which plays a crucial role in promoting referential transparency (when given the same inputs, always produces the same output) and mitigating unintended side effects. Declarative programming languages encourage the use of immutable data structures. Immutable data facilitates reasoning about programs because you can rely on the fact that data does not change during program execution. Since declarative programming focuses on expressing the desired outcome rather than the precise steps to achieve it, immutable data fits well with this paradigm as it allows the program to be more predictable and less error-prone. immutable approaches are quite *analogous*. Mutable data, on the other hand, can be modified or changed after it is created. Mutable data can introduce complexity and potential issues, such as unexpected modifications or race conditions. By expressing the relationship between the application state and the visible interface as a functional dependency, it makes sense that the visible interface would be recreated every time the state changes and therefore does not require mutability. However, most GUI programming is implemented in languages with mutable data structures, so developer effort is required to maintain declarative UI patterns. Vendors emphasize new skills and new ways of programming, as seen in the case of Fluent UI. Although languages may support the optional use of immutable values, e.g., the `final` keyword in Java, it is still up to developers to learn and adopt them, which we discuss in section 5.6. On the other hand, most functional programming GUI toolkits follow declarative approaches such as Fudgets, introduced and developed by Carlsson and Hallgren [25], wxHaskell by Leijen [104], Fruit by Sage [147], Frantk by Courtney and Elliott [32], and many more, but they are generally thin layers on top of non-declarative

frameworks.

Given the main vendors recent intense focus on developing their own declarative UI frameworks, and the dramatic increase in developer interest (as evidenced by StackOverflow data), I believe that the **future is declarative**. I have confidence that the ability for developers to simply describe their intentions and desired outcomes will enable more adaptable and efficient ways of accomplishing GUI programming tasks.

Chapter 5

GUI Toolkits

Are you familiar with the originator of graphical user interfaces? Have you considered the fascinating lineage of GUIs and their associated toolkits? While we will not delve into a comprehensive discussion of GUIs in this thesis, I have included highlights to put this work into its historical context. This chapter will focus on exploring several significant toolkits.

5.1 Short History

Douglas Engelbart[44], the father of graphical user interface (GUI), made significant strides in GUI development through his creation of the On-Line System (NLS). Implemented in the 1960s, NLS revolutionized computer interaction by introducing text-based hyperlinks and introducing an innovative device called the mouse. Notably, Engelbart's groundbreaking 1968 demonstration of NLS, famously dubbed "The Mother of All Demos," showcased the immense potential of GUI technology.

In the following decade, researchers at Xerox PARC expanded upon Engelbart's concepts, with notable contributions from Alan Kay. Kay took GUI advancements a step further by integrating graphics into the interface of the Smalltalk programming language. This breakthrough occurred on the Xerox Alto computer, which debuted in 1973. The graphical user interface presented in Smalltalk went beyond text-based hyperlinks, providing a rich and intuitive visual environment.

The Smalltalk-based GUI system developed at Xerox PARC became a pivotal milestone, serving as the foundation for most modern general-purpose GUIs. This pivotal shift was driven by the pressing need to enhance usability and efficiency, as early text-based command-line interfaces proved to be less user-friendly for the average computer user. By embracing the graphical user interface paradigm, software interfaces became more accessible, intuitive, and visually engaging for a wider audience.

5.1.1 An Overview of GUIs, Widgets, and GUI Toolkits

In the realm of computer systems, a GUI, commonly referred to as GUI (pronounced "gooey"), serves as a computer program that facilitates human-computer interaction through the utilization of graphical control elements, often known as widgets. These widgets encompass various components like buttons, scroll bars, and more. According to Martinez [106], 'the primary objective of a GUI is to enable users to swiftly grasp the functionality and efficiently manipulate the underlying system through the interface'.

As stated by Myers [122], "every widget serves a distinct purpose in facilitating user-computer interaction, manifesting as a visible element within the application's

GUI". Certain widgets enable user interaction, such as labels, buttons, and checkboxes, while others function as containers that group together added widgets, including windows, panels, and tabs. In 1988, the term 'widget' was attested in the context of Project Athena and the X Window System. In an 'Overview of the X Toolkit' by Joel McCormack and Paul Asente, it says: "The toolkit provides a library of user-interface components ('widgets') like text labels, scroll bars, command buttons, and menus; enables programmers to write new widgets; and provides the glue to assemble widgets into a complete user interface" [109]. The reasoning behind the term is indicated by Swick and Ackerman [159] as follows: "We chose this term since all other common terms were overloaded with inappropriate connotations. We offer the observation to the skeptical, however, that the principal realization of a widget is its associated X window and the common initial letter is not un-useful." .

A widget toolkit, widget library, GUI toolkit, or UX library is a library or a collection of libraries containing a set of graphical control elements (i.e., widgets) used to construct the GUI of programs. A toolkit is a specialized library comprising a collection of controls or widgets, including menus, buttons, and scroll bars, among others. These toolkits are designed with a programmatic interface, intended to be utilized by programmers to create user interfaces. Galitz [64] highlights this aspect by stating, "A toolkit is a library of controls or widgets such as menus, buttons, and scroll bars. Toolkits have a programmatic interface and must be used by programmers."

When it comes to developing user interfaces, toolkits play a crucial role by offering a comprehensive range of interactive components and an architectural framework. As stated by Myers et al. [121], "Toolkits typically provide both a library of interactive components, and an architectural framework to manage the operation of interfaces

made up of those components. Employing an established framework and a library of reusable components makes user interface construction much easier than programming interfaces from scratch."

5.2 Imperative Toolkits

Smalltalk

Smalltalk is an object-oriented programming language that emerged in the 1970s with the primary objective of teaching programming to children. Its design focused on creating a language that is small and simple, making it accessible for complete beginners as stated by Eng [51]. Smalltalk boasts a concise syntax and straightforward execution semantics, enabling users to grasp its concepts quickly. One of Smalltalk's distinctive features is its message-passing model, where objects collaborate by exchanging messages. "It was the first computer language based entirely on the notions of objects and messages." as stated by Kreutzer [98]. Unlike other languages, Smalltalk does not incorporate constructors, type declarations, interfaces, or primitive types. Instead, it emphasizes a pure object-oriented approach. What sets Smalltalk apart is its unique self-reflective nature. The entire Smalltalk system, including its compiler, debugger, and programming tools, is implemented in Smalltalk code. This means that users have the ability to read and modify the system's components. This aspect not only promotes a sense of transparency but also empowers both novice programmers, who can easily explore and understand the system, and experienced developers, who can engineer sophisticated solutions. Furthermore, Eng [49] claimed "Smalltalk introduced the Model-View-Controller (MVC) architectural pattern", which has become synonymous with the traditional Smalltalk-80 user interface.

He also said [50], this groundbreaking pattern, characterized by its overlapping windows, has been widely adopted and replicated by operating systems like Macintosh and Windows. Smalltalk was instrumental in developing the graphical user interface (or GUI) and the "what you see is what you get" (WYSIWYG) user interface as mentioned by Porter III [137]. In summary, Smalltalk's emphasis on simplicity, object-oriented principles, and its pioneering MVC pattern has made it a significant language in programming education and professional software development, enabling users to start coding effortlessly while also fostering elegant and adaptable solutions. Squeak and Pharo's toolkits are a part of the Smalltalk programming ecosystem.

Squeak

Squeak, introduced in 1996, is an open-source Smalltalk programming system designed to run efficiently on various platforms. It offers fast execution environments for major operating systems. One of its key features is the Morphic framework, which facilitates the development and maintenance of graphical and interactive applications with minimal effort. Squeak has seen numerous successful projects built upon it, demonstrating its effectiveness as a platform for software development. One notable application of squeak is the original Scratch environment as described by **?**]. Scratch is the world's largest free coding community for kids to program their own interactive stories and games including ScratchJr accroding to Resnick [145], Scratch Foundation [149], an iPad app. This was one of the motivations to create ElmJr, a projectional editor for Elm focussed on our graphics library. Using ElmJr, children transform programs through contextual menus iPad Elm editor.

Morphic is a user interface construction kit that enables direct manipulation of

graphical objects called Morphs. It is built on display trees and serves as a replacement for the original Model View Controller graphics toolkit in Smalltalk-80. The term "Morph" derives from the Greek words for "shape" or "form." In Morphic, a Morph represents the core abstraction. Essentially, a Morph is a Squeak object with a visual representation that can be interactively picked up and moved.

Morphic offers various capabilities for Morphs. They can perform actions in response to user inputs, trigger actions when dropped onto or by another Morph, execute actions at regular intervals, and control the arrangement and size of their submorphs. Additionally, Morphic includes the Morphic Designer, an application designed to simplify the creation of Morphic user interfaces. The Morphic Designer follows the principles of the QtDesigner found in the Nokia Qt Framework, providing a user-friendly environment for designing and crafting Morphic-based interfaces as inferred from Self Language Team [151].

Pharo

Black et al. [17] mentioned in the book that "Pharo is a modern open-source development environment for the classic Smalltalk-80 programming language." The stated goal of Pharo is to revisit Smalltalk's design and enhance it. Pharo originated as a fork of Squeak. It is specifically designed to prioritize simplicity and provide immediate feedback during development. Pharo enhances the capabilities of Squeak, providing a powerful environment with features such as advanced reflection, software-as-objects approach, closures with non-local returns, immediate objects identity swapping, fast resumable exceptions, and easy call stack manipulation, making it versatile and efficient programming [135].

AWT (Abstract Window Toolkit)

Abstract Window Toolkit (AWT) serves as the foundational layer for Swing, Java's graphical user interface framework. Introduced by Sun Microsystems with the initial release of Java in 1995, "AWT widgets provide a lightweight abstraction over the underlying native user interface" as described by Oracle [130]. While AWT prioritizes optimized performance, it may lack certain advanced features. Consequently, AWT is well-suited for smaller Java UI applications that do not require intricate graphical interfaces, making it a preferred choice for full-stack Java developers. AWT's capabilities encompass native user interface components, a robust event-handling model, extensive graphics and imaging tools for shapes, colors, and fonts, versatile layout managers facilitating adaptable window arrangements independent of specific sizes or screen resolutions, as well as data transfer classes enabling seamless cut-and-paste functionality through the native platform clipboard as innferred from Wikipedia contributors [173].

Swing

According to Wikipedia contributors [175], Java Swing (Sun Windowing) is a Graphical User Interface (GUI) toolkit for Java, introduced in 1997 and released in 1998 as part of the Java Foundation Classes (JFC). It offers a wide range of widgets and packages to create sophisticated GUI components. Swing is built on the Java AWT and provides a lightweight approach by rendering its controls using Java 2D APIs instead of relying on native GUI toolkits. This enables Swing components to be platform-independent and highly customizable. Key features of Swing include its flexible and customizable nature, allowing developers to override default implementations and create their own look and feel using the 'LookAndFeel' mechanism. It supports a strong set of widgets and offers built-in support for Undo/Redo functionality. Swing

follows the Model-View-Controller (MVC) design pattern, facilitating a decoupling of data and user interface controls. It also utilizes multi-threading techniques to enhance performance. Swing's configurability and runtime adaptability enable hot-swapping of user interfaces and uniform changes in the look and feel of applications without modifying the code. It simplifies 2D graphics rendering, supports pluggable look and feel, and provides a platform-independent environment. Swing's model-centric approach allows programmers to work with default implementations or create their own models. Overall, Swing provides a rich set of GUI components, extensibility, platform independence, and flexibility, making it a powerful toolkit for Java GUI programming as inferred from the documentation by Oracle [129].

SWT (Standard Widget Toolkit)

SWT, the Standard Widget Toolkit, is a powerful open-source widget toolkit for Java that offers developers efficient and platform-specific access to the user-interface capabilities of various operating systems, as mentioned by Kestermann [94]. It was initially released in 2003 and has since provided developers with a versatile toolkit for creating robust graphical user interfaces. One of the key advantages of SWT is its ability to function independently of the Eclipse Platform, making it a flexible choice for developers working on diverse Java projects as described by Guindon [79]. SWT boasts several notable features that contribute to its appeal. SWT excels in performance, offering faster loading components compared to Swing, another popular Java widget toolkit [126]. Additionally, SWT is designed with optimal memory usage in mind, resulting in smaller memory footprints for applications built with it. A standout feature of SWT is its ability to provide different styles for different types of menus, enhancing the customization options for developers. This allows for greater flexibility

in designing visually appealing and intuitive menus tailored to specific application requirements. SWT and Swing are distinct tools with separate objectives. SWT aims to offer a unified interface for accessing native widgets on various platforms, prioritizing high performance, native appearance, and seamless integration with the underlying platform. On the contrary, Swing is intended to provide a customizable look and feel that is consistent across different platforms. SWT focuses on platform-specific integration and performance, while Swing emphasizes a uniform visual style that can be tailored to individual preferences across all supported platforms as inferred from posts by Eclipsepedia [46], Wikipedia contributors [167].

JavaFX

JavaFX, released in 2008, is a modern GUI toolkit introduced as a successor to Swing. As explained by Pawlan [133], JavaFX is a comprehensive set of graphics and media packages that empowers developers to create cross-platform rich client applications. It offers a range of key features that contribute to its versatility and usability. JavaFX provides Java APIs, allowing developers to leverage the familiar Java language and interact with classes and interfaces written in native Java code. It offers FXML, an XML-based markup language, and Scene Builder for intuitive GUI design. The WebView component enables seamless integration of web pages within JavaFX applications, enabling bi-directional communication between JavaScript and Java APIs as inferred from Oracle [131]. One notable capability is the interoperability with Swing, allowing existing Swing applications to incorporate JavaFX features like advanced graphics, media playback, and embedded web content. JavaFX offers a variety of built-in UI controls, which can be customized using CSS. The Canvas API facilitates direct drawing within the application's scene. JavaFX supports multitouch

operations, follows the Model-View-Controller (MVC) design pattern, and benefits from a hardware-accelerated graphics pipeline (Prism) for fast and smooth rendering. It includes a high-performance media engine based on the GStreamer multimedia framework. The self-contained application deployment model simplifies distribution by packaging all resources and the Java and JavaFX runtimes. These applications can be installed and launched like native applications on different operating systems as inferred from Wikipedia contributors [174].

5.3 Web Development toolkits

UI Kit

This is Apple's original UI framework for building iOS and macOS applications. UIKit is part of CocoaTouch [6], which was released as part of the iOS SDK in 2008, and was available with the first public release of iOS, back then known as iPhoneOS. As mentioned by Jeroen [90], Apple's UI kit was developed based on the Objective-C language, which brings Smalltalk-like object-oriented programming to the C language. It inherited these features from NextStep, as explained by Larkin et al. [102]. UIKit follows the imperative programming style but has introduced more declarative features over time. Developers can define UI components using Interface Builder or programmatically using Swift or Objective-C. UIKit follows the Model-View-Controller (MVC) architectural pattern. UIKit delegates play a crucial role in facilitating communication and data flow between different components of an iOS app. "Delegates are a way for one object to communicate and send data to another object or notify it of certain events" according to the Apple [8] documentation. Delegation allows one object, known as the delegate, to handle specific tasks or provide

information for another object, known as the delegating object. The delegating object typically has a delegate property or a delegate protocol, and it calls specific methods on the delegate to notify or request information. The concept of delegates is not unique to UIKit, it is commonly used in various frameworks and libraries across different programming languages. For instance, in Java Swing, the *ActionListener* interface is a delegate that responds to button clicks and menu selections. Delegates in .NET allow objects to subscribe to and handle events raised by other objects. The *EventHandler* delegate, for example, is commonly used to handle events in Windows Forms applications. Cocoa extensively uses delegates for event handling and object communication. For example, in macOS development, the *NSApplicationDelegate* protocol is used to handle application-level events, such as launching and quitting the application. JavaScript frameworks like React and Angular use a similar concept called "props" and "inputs/outputs," respectively, to achieve a similar effect of passing information and behavior between components.

Angular

AngularJS, released in 2010, was the predecessor to Angular. AngularJS support has officially ended as of January 2022 as reported by Hevery [84] and recommended using Angular according to Hevery [85]. Angular is an application-design framework and development platform for creating efficient, clean, and maintainable single-page applications, two-way data binding, unit testability, reusable components, and support for dependency injection and separation of concerns. Angular 2.0 was a complete rewrite aimed at optimizing the library's compiler. The most recent Angular release builds upon Angular 2.0's foundation with a focus on compiler optimization and speed. Angular follows a component-based architecture, which encourages splitting

code into components, each following the MVC or MVVC design patterns. Angular support for two-way data binding, enabling bidirectional data binding between HTML tags and JavaScript components. Angular offers cross-platform development capabilities, allowing the creation of web, mobile *(Cordova, Ionic, NativeScript)*, and desktop *(Mac, Windows, Linux)* applications. The framework utilizes a Component Router for automatic code-splitting, optimizing speed and performance by loading only relevant code. According to Mukherjee [119] Angular 2.0, released in 2014, introduced a structured approach for projects consisting of modules, components, and services. Each Angular component includes a template view, a class for application logic, and decorators for locating the template view and class. Angular components can form a nested relationship using loops to develop parent-child component relationships. Based on Vyas [163]'s and Manjunath [105]'s blogs, the template view utilizes HTML with two-way data binding, enabling communication between parent and child components through input data and output events.

React and React Native

React is a popular JavaScript UI library that simplifies front-end development by providing support for one-way data binding and a virtual DOM [156]. The history of React dates back to 2011 when it was initially developed as FaxJS by a software engineer at Meta (formerly Facebook). After Meta recognized its potential, the library was rebranded as "React" and implemented in their newsfeed in 2011, followed by Instagram in 2012. React was later open-sourced at JSConf US in May 2013, allowing developers worldwide to benefit from its capabilities. The concept of React Elements, embedded within the library, allows for efficient and real-time updates in the virtual DOM, resulting in improved performance compared to the traditional real

DOM. React components facilitate one-way data binding using "props," enabling the transfer of data from parent to child components. In 2017, React introduced "React Fiber", a set of new algorithms designed to enhance rendering and component compilation. This update focused on improving the speed and optimization of React's core functionalities. React has gained immense popularity due to its efficient rendering, modular component-based architecture, and thriving community that supports it. Developers appreciate React's focus on performance and optimization, making it a go-to choice for building modern and responsive user interfaces, as it's demonstrated in Baer [12]'s book.

React Native is a framework based on React that was initially released by Meta (formerly Facebook) in March 2015 [170]. It enables developers to build native mobile applications for iOS and Android platforms using JavaScript and React concepts. While React is primarily focused on web development, React Native extends React's capabilities to mobile app development. React Native follows a declarative approach, similar to React, where developers describe the desired user interface and the framework handles the underlying rendering and updates. Instead of building separate UI components for each platform, React Native uses a set of pre-built, platform-specific components that are rendered as native elements as shown in documentation by Source [155]. This approach allows for the creation of mobile apps with a native look and feel. By using React Native, developers can write a single codebase that is shared between iOS and Android platforms, reducing development time and effort. React Native also provides a bridge that allows JavaScript code to interact with native components and APIs, enabling access to device features and capabilities. One of the

key advantages of React Native is its ability to achieve near-native performance by leveraging the underlying platform's rendering capabilities. This is accomplished by translating the React Native components to their native equivalents during runtime as described in Ubah [162].

Bootstrap

Bootstrap aims to provide developers with a comprehensive set of tools and components for building responsive and visually appealing web interfaces, as stated by Dykraf [45]. It focuses on simplifying the UI development process and ensuring consistency across different devices and browsers. The first version of Bootstrap was released in 2011. Bootstrap offers a responsive grid system for creating flexible layouts, a collection of pre-styled CSS components (e.g., buttons, forms, navigation bars), and JavaScript components (e.g., carousels, modals) for enhancing interactivity. Bootstrap follows a modular and component-based approach to UI development. Developers can combine and customize Bootstrap's CSS classes and JavaScript components to create the desired UI elements and functionality. It's worth noting that while Bootstrap primarily focuses on UI design and presentation, it can be combined with other frameworks or architectural patterns (such as MVC or MVVM) to achieve a more structured and organized data flow within an application.

Vue.js

"Vue is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model that helps you efficiently develop user interfaces, be they simple or complex" as descibed in [48] and You [178] websites. VueJS was introduced

in 2014 and aimed to create a frontend framework that incorporated the essential aspects of Angular. Notably, VueJS distinguishes itself by offering a beginner-friendly environment and enabling rapid development of small-scale applications with minimal maintenance overhead. Positioned as a progressive framework for single-page applications, VueJS primarily emphasizes the "View" layer in the MVVM design pattern, albeit with some deviations. However, its functionality can be expanded through the integration of third-party packages like Vue Router or Vuex, enabling the utilization of a comprehensive framework's capabilities. Notably, VueJS components are self-contained and can be seamlessly integrated across the application. In recent years, significant efforts have been made to optimize the compiler, resulting in faster rendering and compilation of components, thereby enhancing performance and user experience, as stated by Kofi Group [96].

Material UI

Material UI is a React-based UI framework that implements the Material Design guidelines by Google [107]. The first beta version of Material UI was released in 2017. Material UI aims to provide a set of reusable and customizable UI components that adhere to the principles of Material Design. You can learn more about this from Bernales [16]'s blog. Material UI follows the component-based architecture of React, allowing developers to compose UI elements into a coherent interface. Material UI components are designed to offer graphical indications consistent with the Material Design principles. It provides a wide range of ready-to-use components for building visually appealing and responsive UIs.

5.4 Functional Toolkits

The reactive programming paradigm is based on the synchronous dataflow programming paradigm as indicated by Lee and Messerschmitt [103] but with relaxed real-time constraints. It introduces the notion of behaviors for representing continuous time-varying values and events for representing discrete values. In addition, it allows the structure of the dataflow to be dynamic (i.e., the structure of the dataflow can change over time at runtime) and supports higher-order dataflow (i.e., the reactive primitives are first-class citizens) according to Cooper [31]'s and Sculthorpe [150]'s Ph.D. theses. Most of the research on reactive programming descends from Fran[47, 87], a functional domain-specific language developed in the late 1990s to "ease the construction of graphics and interactive media applications using purely functional and composable approaches". Functional Reactive Programming (FRP), introduced by Elliott and Hudak (1997) is a programming paradigm that combines the principles of functional programming with reactive programming to address complex event-driven systems. In FRP, programs are structured around the concept of time-varying values, represented as streams or signals, which can be transformed and combined using higher-order functions. This allows developers to express and manipulate dynamic behavior in a declarative and composable manner. FRP promotes a clear separation of concerns by enabling the explicit modeling of both the occurrence of events and the behavior of values over time. By providing a more concise and expressive way to handle events and asynchronous data flow, FRP facilitates the development of reactive systems, user interfaces, and interactive applications with increased modularity, reusability, and maintainability as explained in the article written by Bainomugisha et al. [13].

Fudgets Haskell:

As determined by Carlsson and Hallgren [26], Fudgets is a small window-based graphical user interface toolkit for X Windows written in the lazy functional language Lazy ML(LML) between 1991 and 1996 and the work continued for several years including the recent changes that was released in 2016 as reported by Hallgren and Carlsson [81]. It is also one of the most well-known functional models for GUI programming in Haskell. Its primary goal is to provide a 'purely functional' (referring to Okasaki [128]) and declarative approach to building graphical user interfaces. The extensive Fudget library uses X windows and is supported by many Haskell compilers. Fokker et al. [57]. Fudgets is the abbreviation of 'functional widget', whereas widget is an abbreviation of 'window gadget'. Fudgets are composable functional widgets that can be combined to create complex UIs as claimed by Carlsson and Hallgren [25]. The toolkit follows the FRP paradigm, where UI components react to changes in input signals. The architecture used in Fudgets Haskell is typically based on a hierarchical structure of fudgets. The Fudget combinators give a rigid structure to the data flow in a program. In functional programming, a combinator is a higher-order function that combines two or more functions to produce a new function. Combinators are used to build more complex functions from simpler ones, and they are often used in functional programming libraries to provide a concise and expressive way of defining behavior. In Fudgets, combinators are used to define the appearance and behavior of widgets in a Graphical User Interface (GUI). The visual representation of the data flow through its infix operator. This operator creates a clear pipeline-like structure, allowing developers to easily understand how signals and data are transformed and

propagated between GUI components. The visual nature of the infix operator enhances the readability and comprehension of the code. The fudget concept has been implemented on top of a number of GUI toolkits on a popular GUI library called Gadget, in accordance with Noble and Runciman [124]'s thesis, where Gadget stands for 'generalized fudget'. The author says the motivation for this name is that "gadgets are processes that communicate via typed, asynchronous channels (called wires), thus allowing a gadget to have an arbitrary number of input and output pins".

WxHaskell

WxHaskell is a GUI toolkit for Haskell that provides bindings to the wxWidgets library. Its main goal is to enable Haskell programmers to build native-looking and platform-independent GUI applications. WxHaskell follows an imperative programming style, similar to how GUI applications are typically built in languages like C++ with wxWidgets. WxWidgets provides a common interface to native widgets on all major GUI platforms, including Windows, Gtk, and Mac OS X. It has been in development since 1992 and has a very active development community and the last updated version of wxHaskell is from 2021. The objective of wxHaskell, as presented in the paper by Leijen [104], is to demonstrate the use of mutable variables for communication between event handlers. The paper acknowledges the extensive research focused on avoiding mutable states and promoting a declarative approach to GUI programming. However, since this remains an active area of research, the authors chose to prioritize the creation of a standard monadic interface for the library as their initial goal.

Frantk

FranTk is a high-level library for programming Graphical User Interfaces (GUIs)

in Haskell released in the year 2000. It is based on Fran (Functional Reactive Animation), which was released in the year 1997. It uses the notions of Behaviors and Events to structure code. FranTk allows a compositional, declarative style of programming with both static and dynamic user interfaces. In order to provide a powerful set of platform-independent set of widgets, FranTK uses binding to the popular Tcl/Tk toolkit. It allows for building web applications with a functional and type-safe approach. It provides a seamless integration of Haskell and web technologies. As specified by Elliott and Hudak [47], Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of behaviors and events. It uses behaviors to represent the state of an application and events to represent user input. They both are first-class values. Behaviors are time-varying, reactive values, while events are streams of values that occur over time. Frantk follows a FRP paradigm and leverages the power of type-level programming to ensure type safety and correctness in web applications as described by Sage [147].

Reflex

Reflex is a fully-deterministic, higher-order FRP interface and an engine that efficiently implements that interface in Haskell released in 2006, as mentioned by Trinkle [161]. It aims to provide a high-level and composable approach to UI development. Reflex follows the FRP paradigm, where UI components are defined in terms of behavior and events. It leverages the concept of dynamic values that can change over time. Reflex is a library that serves as the foundation for FRP, consisting of three primary types: behavior, event, and dynamic. Behavior represents a value that can change over time and can be sampled at any point but does not provide notifications

when it changes. It abstracts the concept of a value existing at all time points. Event captures discrete occurrences or updates at specific time points. It is push-oriented, informing you when the value changes, and can represent events such as button clicks or key presses. Dynamic combines the characteristics of both event and behavior. It holds a value at all time points and can notify you when its value is updated. It can be seen as a step function over time, encompassing an event and a behavior. Reactive programming in Reflex involves utilizing various sources of events to provide responses. Instead of using explicit callbacks or function calls, responses are expressed by firing another event or modifying a dynamic value. These event and dynamic values propagate through widgets, allowing them to respond appropriately to events. The propagation forms an event propagation graph, potentially creating cyclic dependencies, based on the documentation written by Trinkle [160].

Fruit

Fruit is a Functional Reactive User Interface Toolkit, a graphical user interface library for Haskell, based on a formal model of user interfaces. Its implementation is in-progress and began in 2018. As inferred from the paper by Courtney and Elliott [32], Fruit is based on a formal model of user interfaces that identify signals (continuous time-varying values) and signal transformers (pure functions mapping signals to signals) as core abstractions. The model defines GUIs compositionally as signal transformers, which means that GUIs are built by combining simpler signal transformers into more complex ones. This approach allows for a high degree of modularity and reusability in GUI design. Signals are continuous time-varying values, and signal transformers are pure functions that map signals to signals. In other words, a signal is a function that takes time as an input and produces a value as

an output, while a signal transformer is a function that takes one or more signals as inputs and produces one or more signals as outputs. Signal transformers can be used to modify or combine signals in various ways to create more complex behaviors in a system. States in Fruit are based on Arrows-based Functional Reactive Programming (AFRP), its FRP, based on Arrows i.e. is a generalization of Monads, is an adaptation of ideas from Fran and FRP to the arrows framework proposed by Hughes. Both Fran and FRP are known for their use of monads to model time-varying computations. This approach is similar to the Model-View-Controller (MVC) design pattern in Fruit, signals represent data that changes over time, signal transformers represent logic that transforms those signals into new signals, and GUIs are composed of signal transformers that define their appearance and behavior. This approach allows for a clear separation of concerns between different parts of the application and promotes modularity and reusability.

Fruit Haskell follows a functional programming style and provides a minimalistic set of UI components. The authors of the paper, Courtney and Elliott [32], acknowledge the challenge of balancing expressiveness and simplicity in the design of the Fruit Library. They aim to provide a powerful and flexible library for GUI programming while ensuring it remains accessible to developers unfamiliar with functional programming concepts. Another challenge is optimizing performance while maintaining modularity and composability in signal transformer composition. The authors suggest future work should focus on developing techniques to optimize performance without compromising functional purity. Additionally, they highlight the need for further research in exploring the full potential of their approach, including developing more sophisticated examples and applications and extending support for

advanced features like animation, layout management, and event handling.

The following have their own graphical builders: **WxHaskell:** WxHaskell provides a visual designer tool that allows developers to create and modify GUI components visually. This graphical builder enables developers to design the user interface by visually arranging and configuring widgets. **Gtk:** Gtk offers visual layout designers that allow developers to visually design and arrange GUI components. These graphical builders provide a user-friendly interface for creating and modifying the application's user interface.

5.5 Declarative Toolkits

Flutter

Flutter is a cross-platform UI toolkit developed by Google and released in 2017 [168]. As inferred from documentation by Google [71], Flutter uses the Dart programming language and features a widget-based system, where widgets represent different parts of the UI. Flutter is a reactive, *pseudo-declarative UI framework* as described in [70], in which the developer provides a mapping from the application state to the interface state, and the framework takes on the task of updating the interface at runtime when the application state changes. Flutter aims to enable cross-platform development with a single codebase, allowing developers to build visually appealing and performant applications. It emphasizes declarative UI as discussed in 4.1.2. Flutter categorizes all its user interface elements as 'widgets,' includes controls, containers, and layout components. It offers both stateless and stateful widgets to support different programming needs. While the official documentation lists 206 widgets and

823 classes for multi-platform support, the exact number of widgets can vary as inferred from Google [69]. Some sources, such as a YouTube video by FlutterMapp [56], mention 215 widgets in Flutter.

Among these widgets, there are multiple variations of the same control, such as buttons, designed to serve specific purposes or styles. Examples include Material Button, Outlined Button, Text Button, Toggle Buttons, Icon Button, Dropdown Button, or Elevated Button. While these variations are referred to as individual 'widgets,' they possess different properties while belonging to the same widget category. Although Flutter attempts to mimic native platform widgets, it may not provide an exact match for every widget. For instance, the UIKit's UIStepper and its counterpart in Flutter, 'CupertinoStepper,' have differences in functionality and purpose.

In UIKit, 'a stepper is a two-segment control primarily used to increase or decrease an incremental value'. On the other hand, the 'Material Stepper in Flutter is a widget designed to display progress through a sequence of steps' as per documentation by Pub.dev [140]. Steppers are particularly useful in scenarios where one step depends on the completion of another or when multiple steps must be completed to submit a form.

SwiftUI

Worldwide Developers Conference (WWDC) 2019, Apple announced a new framework called SwiftUI for building user interfaces across all Apple platforms. It is Apple's modern UI framework for building applications on iOS, macOS, watchOS, and tvOS. SwiftUI introduces a declarative syntax that enables developers to describe UI components and their behavior concisely. It follows the Model-View-ViewModel (MVVM) architectural pattern and leverages Swift's language features for seamless

integration with the underlying platform as inferred from the documentation by Apple [7]. In addition, Davis [41] reported that "Aiming to decrease the lines of code, SwiftUI supports declarative syntax, design tools, and live editing". SwiftUI aims to simplify UI development for Apple platforms by providing a declarative and intuitive approach, based on Barker [14]'s book. Destin mentions that "It allows developers to build responsive and visually appealing user interfaces with less code." and he also, exhibits a practical example in this handbook [43].

Fluent UI

Fluent UI is a UI toolkit developed by *Microsoft* for creating web and desktop applications [112]. Fluent UI provides a set of reusable components and styles that follow *Microsoft's Fluent Design System* a design language developed in 2017 by Microsoft, as explained by Clarke and Gusmorino [27]. Fluent UI is the 2020 new name for *UI Fabric*[111] of Microsoft's Fluent Design System. It supports declarative and imperative programming paradigms, allowing developers to choose their preferred approach. It aims to provide a unified and customizable UI toolkit for developers, as claimed by McLaughlin [110]. Fluent UI does not enforce a specific architectural pattern and can be used with various frameworks and patterns, such as component-based architectures or MVC. However, as discussed in this Microsoft Fluent Wiki [113], it strongly recommends describing user interfaces using the declarative approach.

Android Jetpack Compose

Jetpack Compose [73] is a modern declarative UI Toolkit for Android [68]. It leverages the power of Kotlin's language features to provide a concise and expressive way of defining UIs. It was launched in 2021 and is now stable and ready for adoption in production, as claimed in Bellini and Butcher [15]'s blog post. Jetpack Compose

is built on top of the Android framework and follows the Model-View-ViewModel (MVVM) architectural pattern. "Jetpack aims to simplify UI development for Android apps by using a declarative approach. This UI Toolkit focuses on providing a modern and efficient way to build user interfaces with less boilerplate code" as inferred from Google [72]

Elm-UI

Czaplicki and Chong [38] in the section "Building GUIs with Elm" says "Elm's purely functional and declarative approach to graphical layout, allows a programmer to say what they want to display, without specifying how this should be done." Elm-UI is a library for building user interfaces in Elm programming and was published in 2020. It is designed to be a high-level design toolkit that draws inspiration from the domains of design, layout, and typography, as opposed to drawing inspiration from HTML and CSS like most other UI libraries. Elm-UI is purely functional and emphasizes usability, performance, and robustness as inferred from Griffith [78]. Elm-UI provides a declarative syntax for building user interfaces, which allows developers to describe the layout of their applications in a clear and concise manner. It uses a virtual DOM approach to make updates efficient [33]. Elm-UI is built on top of Elm's core architecture, which is a pattern for building interactive web applications. An Elm program is always split into three parts: Model, View, and Update. The 'Model' represents the state of the application, the 'View' is a function that turns the Model into HTML, and the 'Update' function handles user input and updates the Model accordingly as referred by Korban [97].

5.6 Common Challenges and Solutions for Immutability Adoption

Why do we have so many toolkits? What is the common issue that all these toolkits are trying to solve?

The existence of multiple GUI toolkits stems from the historical evolution of graphical interfaces, as advancements in technology and the emergence of new platforms necessitated the development of specialized toolkits. This historical context contributes to the diverse range of GUI toolkits available today, empowering developers to choose the most suitable toolkit for their specific project requirements and preferences. The existence toolkits provide a standardized and platform-independent approach to graphical interface development, allowing applications to run seamlessly on different operating systems with distinct rendering mechanisms. The introduction and creation of new programming languages, for instance, the *Dart* programming language was indeed created as a part of the Flutter framework(4.1.2). Initially, it was intended as a general-purpose programming language, but it gained significant prominence as the primary language for building applications using the Flutter framework. It helps to reduce the complexity of user interface design, and the desire to separate business logic from interface concerns and its underlying complexity. Therefore, various programming language offers language-specific bindings or APIs, enabling developers to leverage the capabilities of their preferred programming languages and ensuring compatibility with different language ecosystems. Moreover, these toolkits often target specific use cases such as desktop applications, mobile app development, or embedded systems, providing features, libraries, and performance optimizations

tailored to each domain. This led to the proliferation of multiple toolkits in the field attributed to a variety of factors and considerations.

In "The Ultimate GUI Framework: Are We There Yet?" [158], Stokke et al discuss the landscape of modern GUI frameworks and their different approaches to separating the data layer from the presentation layer. "The common task of all these frameworks is to keep an application's view in sync with its model, and that framework that updates the DOM automatically on model changes is called reactive." It also describes how these frameworks update views and track variable changes and provides a cross-tabulation of the properties and frameworks. The properties include declarative view specification, re-rendering mechanism, two-way bindings, stateful components, component hierarchy, multi-way dataflow, and more. Furthermore, the authors of this research paper say "despite the numerous advancements in GUI programming since the inception of the Model-View-Controller pattern, **there are still areas that can be enhanced and improved upon** to achieve an ambitious objective of "The ultimate GUI framework," although it may be a challenging goal".

In imperative programming languages, there is a significant emphasis on state mutation, where the values of data structures can be modified throughout the execution of a program. However, to enhance program reasoning and maintainability, language designers have introduced features that restrict the extent of such modifications to data structures. These features aim to impose certain constraints on state mutation, thereby promoting a more controlled and predictable program behavior. By limiting the scope of mutations, developers can gain better insights into program execution and facilitate more effective debugging and maintenance processes, as reported by

Abelson and Sussman [1]. The authors of the research paper titled "Exploring Language Support for Immutability", Coblenz et al. [28], present the design of a novel language extension aimed at facilitating the specification of immutability in Java. In the context of Java programming, the use of constants in combination with the final keyword is commonly employed to enforce immutability. The paper explores the introduction of a language-level extension to further enhance the ability to declare and manage immutable entities within Java programs. If the Java compiler isn't convinced that your final variable will only be assigned once at runtime, then it will produce a compiler error. The 'final' keyword gives you static checking for immutable references, as marked by Max Goldman [108]. Immutable classes greatly simplify programming, program maintenance, and reasoning about programs. As Bugayenko and Zykov state, With compiler support, they argue that immutable classes can be freely shared, even between concurrent threads and with untrusted code. Immutability is a recommended coding practice for Java [21]. The paper also brilliantly explains the immutability adapted by various programming languages. Although *'final'* in Java requires that a particular field cannot be reassigned to refer to a different object, the contents of the referenced object may still change. Zibin et al [179] presents Immutability Generic Java (IGJ), a java annotation that implements immutability. For example, @Immutable Date d is a reference to an immutable date. No fields can be modified on an *@Immutable* object; *@Readonly* in IGJ specifies a read-only reference. Guava a set of core Java libraries from Google that includes new collection types (such as multimap and multiset), immutable collections says "When you don't expect to modify a collection, or expect a collection to remain constant, it's a good practice to defensively copy it into an immutable collection" as described in the

wiki of Google/Guava [77] and all immutable collection implementations are more memory-efficient than their mutable siblings, as stated by Andreou [4].

There are indeed many researchers and authors who advocate for the use of immutability in programming.

- **Rich Hickey**, the creator of the Clojure programming language, has emphasized the importance of immutability in his talks and writings. His "Effective Programs" series of talks, particularly the talk titled "Simple Made Easy," delves into the advantages of immutable data and its impact on program correctness and reasoning Rich [146].

- **John Hughes**, a prominent computer scientist known for his work on functional programming and testing, has highlighted the benefits of immutability in his research. His paper "Why Functional Programming Matters" discusses the safety advantages of immutable data structures and how they can simplify program understanding and debugging Hughes [89].

- **Simon Peyton Jones**, a co-creator of the Haskell programming language and a researcher at Microsoft Research, has written extensively about functional programming and immutability. His papers and talks often discuss the benefits of immutable data and its impact on program correctness and reasoning. One of his contribution in a journal by Hudak et al. [87].

- **Martin Odersky**, the creator of the Scala programming language, has also acknowledged the advantages of immutability. His book "Programming in Scala" and various talks on functional programming emphasize the safety and concurrency benefits of immutability Odersky et al. [127].

The research conducted by Philipp Haller and Ludvig Axelsson titled "Quantifying and Explaining Immutability in Scala" highlights the significance of immutability as a crucial characteristic of data types, particularly in concurrent and distributed programming scenarios. The paper emphasizes how immutability enables the utilization of efficient techniques to ensure fault tolerance in distributed systems. Furthermore, the study delves into an empirical analysis of medium-to-large open-source Scala code bases, aiming to quantify the prevalence of immutability in real-world Scala projects [80].

In a language like JavaScript where immutability is not built into the language, producing a new state from the previous one is a boring, boiler-platy task. Immutable.js written by Lee Byron is a library embracing immutability and enables Javascript's future it provides immutable data structures to make working with complex data easier and more efficient. It offers collections such as List, Map, and Set, which are immutable, meaning they cannot be changed after creation. Instead of modifying data directly, according to Byron [23], Immutable.js provides methods that return new copies of the data with the desired changes applied [24]. Immer.js is a tiny JavaScript library written by Michel Weststrate whose stated mission is to allow you "to work with immutable state in a more convenient way" [165].

The below table (5.1,5.2) shows some existing systems and discusses the type of restriction, scope, transitivity, initialization, abstract vs. concrete State, backward compatibility, enforcement, and polymorphism. Keeping SoC in mind, various programming languages and frameworks have introduced immutability to maintain the application state and avoid unknown states resulting from side effects as it leads to unknown states by this way we can understand the importance of it.

Table 1: Summary of Some Existing Systems (abbreviations are from Table 2)

System	Type	Scope	Trans.	Init.	Abstr.	Compat.	Enforcement	Polymorph.
Java `final`	a	o	n	e	N/A	c	s	n
C++ `const`	r	o	n	e	a	c	s[1]	n
Obj-C immutable collections	i	o	n	e	N/A	c	s[1]	n
.NET Freezable [23]	i	o	n	e	N/A	o	d	n
Java `unmodifiableList`	r	o	n	e	N/A	o	d	n
Guava `ImmutableCollection`	i	o	n	e	N/A	o	s, d[2]	n
IGJ [44]	i, r	c, o	n	e	a	c	s	p
JAC [20]	r	o	t	e	c	c	s	n
Javari [40]	r	c, o	t	e	a	c	s	p
OIGJ [45]	i, r, o	c, o	n	r	a	c	s	p
`immutable` [17]	i, r, o	c, o	t	r	a	o	s	p
C# isolation extension [14]	i, r, o	c, o	t	r	a	c	s	p
JavaScript `Object.freeze`	i	o	n	e	c	o	d	n

[1] These approaches provide static enforcement to the extent possible in these languages.
[2] Static deprecation warning, runtime exception

Figure 5.1: Overview of Language Support for Immutability by Coblenz et al. [28]

Table 2: Summary of Dimensions

Dimension	Possible choices
Type	**i**mmutability, **r**ead-only restriction, **a**ssignability, **o**wnership
Scope	**o**bject-based, **c**lass-based
Transitivity	**t**ransitive, **n**on-transitive
Initialization	**r**elaxed, **e**nforced
Abstraction	**a**bstract, **c**oncrete
Backward compat.	**o**pen-world, **c**losed-world
Enforcement	**s**tatic, **d**ynamic
Polymorphism	**p**olymorphic, **n**on-polymorphic

Figure 5.2: Summary of Dimensions by Coblenz et al. [28]

In the GUI programming context, the Flutter framework introduces 'Stateless Widgets'. The concept of a stateless widget in Flutter refers to a widget whose properties are immutable, and any changes to those properties require creating a new instance of the widget. This is evident from the structure of stateless widgets, as they typically have only one class that extends the StatelessWidget class. Consequently, the build() method of a stateless widget is not re-invoked once it is initially rendered.

Stateless widgets are designed to have only final fields, without any exceptions. This design choice stems from the fact that when the parent widget rebuilds, such as during screen rotation, animations, or scrolling, the build method of the parent is called, leading to the reconstruction of all widgets. Therefore, maintaining final fields ensures that the appearance and properties of a stateless widget remain unchanged throughout its lifetime as described by Woka [176].

According to the documentation Google, Inc. [75], Flutter Agency [55], in Flutter, "a stateless widget cannot alter its state during the runtime of the application". This means that it remains static and cannot be redrawn or updated while the app is in action. Once a stateless widget is initialized, its class is only called once. Even if external factors exert influence, the widget will not be updated. Consequently, the only way to modify a stateless widget is to delete it and create a new instance with the desired changes.

In Dart, the '@immutable' annotation is used to enforce that every field within a class, including its subclasses, must be declared as 'final'. If any field lacks the 'final' keyword, the Dart compiler will issue a warning, but not an error. On the other hand, using the 'final' keyword explicitly denotes that a property cannot be assigned a new value after its initialization, and the Dart compiler will emit an error if such an attempt is made.

To create an unmodifiable list in Dart, you can utilize the 'UnmodifiableListView' class. It acts as a wrapper around a list and prohibits modifications such as adding or removing items. Although it exposes methods like 'add()' or 'addAll()', these methods will throw exceptions at runtime if invoked.

While these approaches provide some level of immutability and protection against

modifications, they do not fulfill the requirements for compiler-time safety. Additionally, the lack of built-in methods for equality, hashing, and cloning in Dart's immutability mechanisms can be limited to creating robust data classes as described by Muccinelli [118], Boformer [19],

Unlike object-oriented programming, instead of relying on inheritance, Elm encourages the use of composition and function composition to achieve code reuse and modularity. Composition involves combining smaller functions or components to build larger ones, promoting a more modular and maintainable codebase. By breaking down functionality into smaller, composable units, developers can create flexible and reusable code that can be easily tested and reasoned about. Pierce [136] mentions that **In the context of Elm**, addressing the capturing of component types involves the use of parametric polymorphism [83], which enables the creation of generic functions or data types that can operate on multiple types. By leveraging this feature, the Elm framework can define components that are parameterized by types, allowing for a more flexible and reusable design.

The capturing of component types becomes particularly crucial when dealing with composite or complex user interfaces. When combining multiple components side by side, it is essential to capture the types of individual components within the overall structure. This ensures that the types align correctly and enforces constraints, preventing runtime errors and enhancing the reliability of the interface.

Immutability plays a significant role in Elm and aids in reasoning about program behavior. In Elm, all properties are declared as final, meaning they cannot be modified after being assigned a value. This emphasis on immutability promotes easier comprehension of the code and ensures more predictable outcomes. Developers can

rely on the immutability of properties and avoid the need to check for mutable states, resulting in a clearer understanding and more robust code. In the next chapter 6, we will discuss our ideas and implementation about having a pure immutable UI toolkit.

5.6.1 Is state a problem?

State is often considered problematic in programming languages, especially when it comes to functions that have side effects. Flutter offers stateless widgets. It is easy to get the impression that state should be avoided. However, without state, there would be no meaningful programs! For example, we are concerned about the state of our bank accounts, and we expect accurate updates when we deposit or withdraw money. Given the existence of state in the real world, programming languages need to provide facilities to handle it. Different approaches have been taken by various language paradigms:

OO Programming Languages (OOPLs) suggest "hiding the state from the programmer." They achieve this by encapsulating state within objects and allowing access to it only through defined methods. The state remains hidden from direct manipulation.

Imperative programming languages like C and Pascal control the visibility of state variables through the scope rules of the language. This means that the availability and visibility of state variables are determined by the specific rules governing the scope in which they are defined.

Pure declarative languages take a different stance and claim that there is no state at all. These languages focus on expressing computations through logical or functional constructs without any notion of a mutable state.

In some functional programming and logic languages, mechanisms like monads and

definite clause grammars are used to hide state from the programmer. These mechanisms allow programming "as if state didn't matter" while still providing limited access to the system's state when necessary.

However, the choice made by OOPLs to "hide the state from the programmer" is often seen as the worst possible option. Instead of exposing the state and finding ways to minimize its drawbacks, OOPLs opt to conceal it entirely, potentially leading to more complexities and limitations in dealing with state-related issues. Although similar criticisms do not exist for functional languages, the sheer volume of monad tutorials indicates that this abstraction is a barrier to beginners.

By building a wrapper around a low-level toolkit that uses mutable data, mainstream Elm programming avoids both of these issues, but is exposed to bugs in the wrapped frameworks, which become impossible to understand in the high-level code. We have experienced this with Elm+Bootstrap.

Can state be exposed in a safe way? This is the motivation for the next chapter.

Chapter 6

A User Interface Toolkit without Mutable Data

> "A programming language is low level when its programs require attention to the irrelevant." –Alan Perlis [134]

> "The last thing you wanted any programmer to do is to mess with *internal state* even if presented figuratively. It is unfortunate that much of what is called 'object-oriented programming' today is simply old-style programming with fancier constructs." - Alan Kay [92]

Taken together, we interpret these quotes as saying two giants of the field believed that low-level programming, in which the programmer manipulates internal state, was a problem, and that programmers were persisting in doing so even when presented with tools created to help them with abstraction. Note that Alan Kay was the creator of Smalltalk and a pioneer in object-oriented programming! Accepting that programmers will do everything we don't want them to do if we don't make

it impossible, we can ask the question: how far can we get in implementing a GUI toolkit using purely immutable data? Using immutable data does not guarantee the programmer will not devote attention to the irrelevant, but that is because this is a judgement call. Using immutable data does prevent external modules from accessing the internal state of other modules through indirect means, such as indirect references which are hard to track down.

On the other hand, we choose to ignore Daan Leijen, an expert in functional programming and developer of wxHaskell, who said

> "We have learned an important lesson from wxHaskell: do not write your own GUI library!" -Daan Leijen [104]

While we agree that wrapping an existing toolkit is faster, and allows for experimentation with the library interface, hiding a lot of the internal state in the toolkit widgets, that state is still there, and it can still be mutated unintentionally, no matter what paradigm the wrapper library exposes, thus opening up a hole through which bugs can crawl.

We feel it is worth answering the question, can we build a GUI Toolkit entirely without mutatable data, whether explicitly in the API, or hidden behind a façade? It is even more urgent now than when wxHaskell was developed because all major vendors and many developers are embracing declarative UIs. Are they fully benefiting from declarative programming if they only use it at the interface to their libraries, and not internally?

This chapter explores the barriers associated with constructing an entire GUI Toolkit, encompassing not only the exposed interface but also the underlying components, using immutable data structures. To achieve this, we sought to adapt the

model-view-update pattern to develop enough widgets, ranging from buttons to sliders, to identify the difficulties which would arise in developing a complete library. We theorized that the resulting library would greatly benefit from the enhanced software quality attributed to immutable data types. We then examined whether this approach would remain accessible to individuals familiar only with object-oriented (OO) GUI toolkits.

This experiment can be conducted in multiple programming languages, particularly those that support modern paradigms. In languages that allow multiple paradigms, we would need to ensure that no mutable data structures are employed directly or indirectly through object usage. However, by utilizing Elm as the implementation language, such verification is unnecessary, as Elm does not incorporate monads or any hidden mutable data structures. The absence of monadic constructs aligns with Elm's functional programming paradigm, emphasizing immutability and explicitness throughout its design. By conducting this research, we aim to shed light on the feasibility and potential benefits of employing immutable data structures for building comprehensive GUI Toolkits.

6.0.1 Elm

Elm, a functional language that compiles JavaScript, was the first mainstream example of an immutable architecture for front-end web programming. The fact that this same architecture was transferred, via Redux, to the mainstream JavaScript community is really fantastic [29]. Elm is a pure language by design, meaning that our code does not have any side effects, instead, it has an explicit state (called the `model`) inputs and outputs. Elm was originally built around functional reactive programming, but

as of version 0.17, functional reactive programming features were removed in favor of model-view-update, also called The Elm Architecture (TEA), with subscriptions [36]. We have discussed this architecture in-depth in section 3.5.

The Elm runtime system handles side effects for us. In our code, we simply request these side effects to be performed and wait for the results. Side effects can include actions like making HTTP requests and receiving responses.

Elm offers two approaches to incorporate what would be side effects in other languages. Firstly, most side-effects in other systems are the main effect. The function `update: Msg -> Model -> (Model, Cmd msg)` *explicitly* modifies the state. The inputs are the `Msg`, a data type that encodes every event in an algebraic (union) data type[1], and the application state—the `Model`. Secondly, for tasks like HTTP requests, we utilize commands (`Cmd`). In some languages, these would be function calls, or remote procedure calls, rather than data.

There is a famous white paper about the 'Software Crisis' by Moseley and Marks [114] references " 'State' as the number one contributor to software complexity". When working with immutable data, making a change to a specific piece of data requires creating a copy of that data with the desired edit applied. However, the new data can still reference the old data. The program can use both versions of the data, and the old data will only be garbage collected when there are no references to it.

Structural sharing refers to a technique used in immutable data structures. When a change is made to a specific part of an immutable data structure, instead of modifying the data in place, a new copy of the structure is created with the desired change applied. However, the new copy still shares most of its memory with the original

[1]When we want to create our own custom types in Elm we use algebraic data types (ADTs). An ADT is a type that is composed of other types. This allows us to define a type and specify all the instances the type can assume.

structure, as only the modified parts are duplicated. This sharing of memory between the old and new structures allows them to coexist and be used simultaneously, facilitating features like easy undos, rollbacks, and efficient memory usage. If the old structure is no longer needed, it can be garbage collected. Modern JavaScript engines have efficient garbage collectors, making the process fast. Structural sharing ensures immutability while minimizing the need for excessive memory allocation and copying of data as inferred from Okasaki [128], ReactEurope [143].

Today, even ostensibly declarative toolkits are often built by encapsulating a lower-level toolkit that employs mutable data structures. Also, in Elm libraries, the presence of mutable data is primarily observed within HTML widgets and the utilization of CSS. To address this concern, an alternative approach involving GraphicsSVG and immutable data is advocated, aiming to avoid mutable states and promote immutable states.

6.0.2 GraphicsSVG

Anand and Schankula [3] were inspired by the original Elm Graphics module, which targeted HTML canvas elements, to create GraphicSVG, which is partially backwards compatible. GraphicSVG's principal types are `Stencil`, `Shape`, and `Collage`, which model real-world concepts, the `Collage` type represents the drawable surface of the window which contains a (x, y) pair of horizontal and vertical dimensions (arbitrary units, not necessarily in pixels) to which the drawing surface will be scaled, and the `List` of `Shape`s to be drawn on the drawing surface. `Stencil` describes a stencil which must be filled or outlined to create a `Shape`. In Figure 6.1, the combinatorial choices are illustrated by the position of a line of highlighting. The architecture of

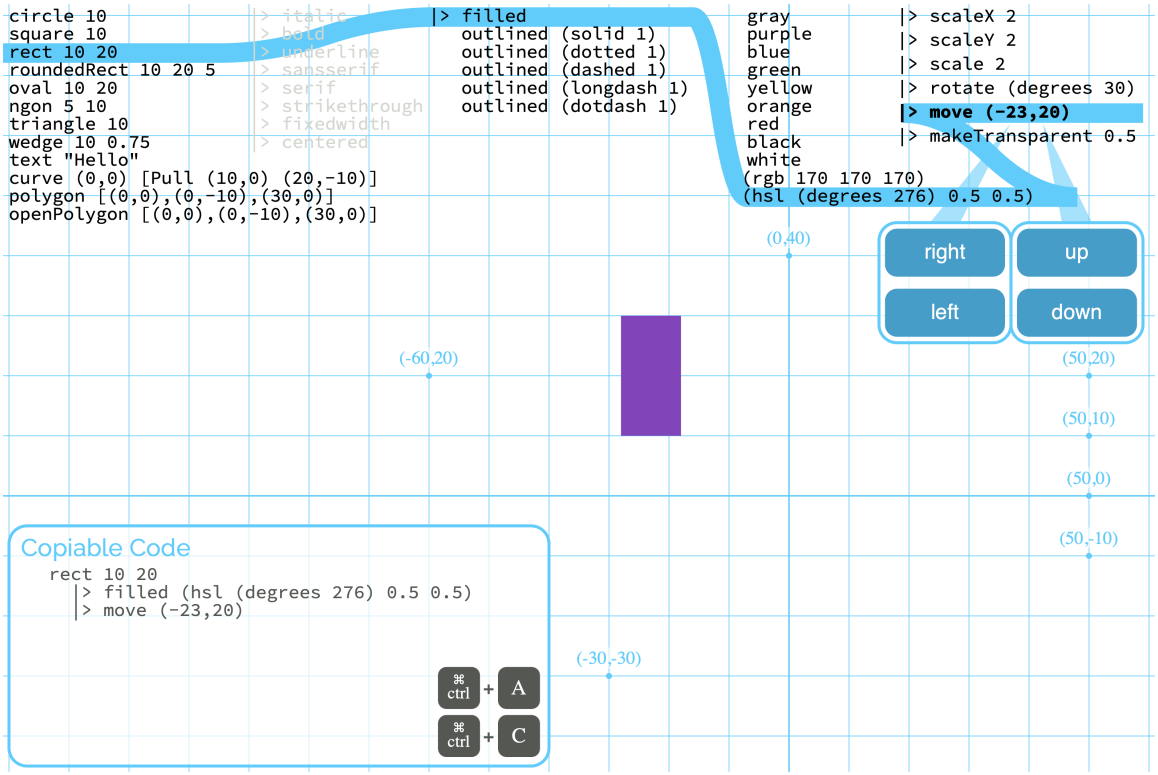

Figure 6.1: The ShapeCreator [58] illustrates the combinatorial structure involved in constructing `Shape`s.

GraphicSVG is designed in such a way that it limits the number of parameters each function takes, by using function composition (`|>`), making them easier to use. This means that you typically pass fewer parameters compared to other graphics libraries or frameworks, and wherever possible the functions are composable. For example, a `Stencil` can be `filled` or `outlined`. The goal is to simplify the process of creating and manipulating vector graphics, especially for beginners and students. GraphicSVG is a powerful tool to construct vector graphics, animations, and interactive programs as described by d'Alves et al. [40].

6.1 A High-Level Overview of the Toolkit

The module hierarchy in Figure 6.2 demonstrates the structured organization of views and their corresponding types. Each of the widgets imports its `Widget` types from the `Types` module thereby it is built around a central type. In order to enhance the clarity of the subsequent explanations we will commence with an illustrative example of a `ToggleView` before delving into a composite view, namely the `SideBySideView`. This approach aims to enhance the understanding of the subsequent discussion and facilitate comprehension of the hierarchical relationships within the system.

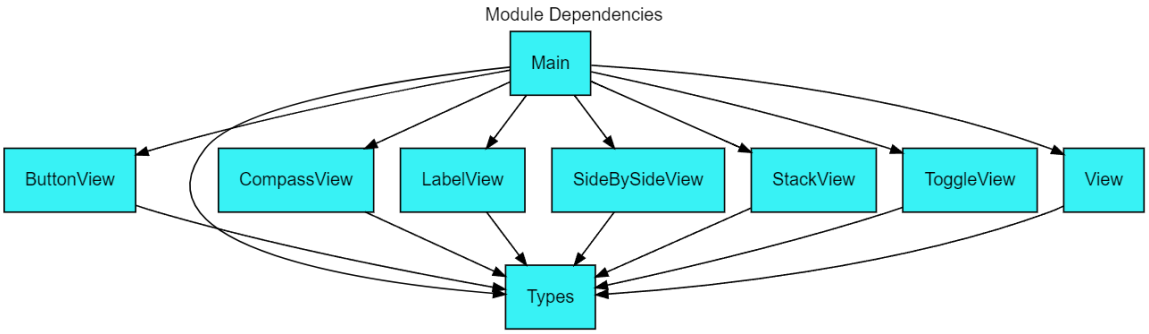

Figure 6.2: Module dependency

6.1.1 MVU with Composability

In Figure 6.3 we expand on the simple MVU dataflow diagram of Figure 3.4 seen previously with an exploded view of the `update` function for an app using our widget library. In the exploded view, we see that the state component of the "business logic" is contained in the top-level model as field of type `:AppState` (a type defined in each application), as is the "widget-tree" state as a field of type `:wState`. But the widget-tree state is contained inside a record of type `Widget wMsg wState msgToApp msgToWidget` as defined in sub-section 6.1.2.

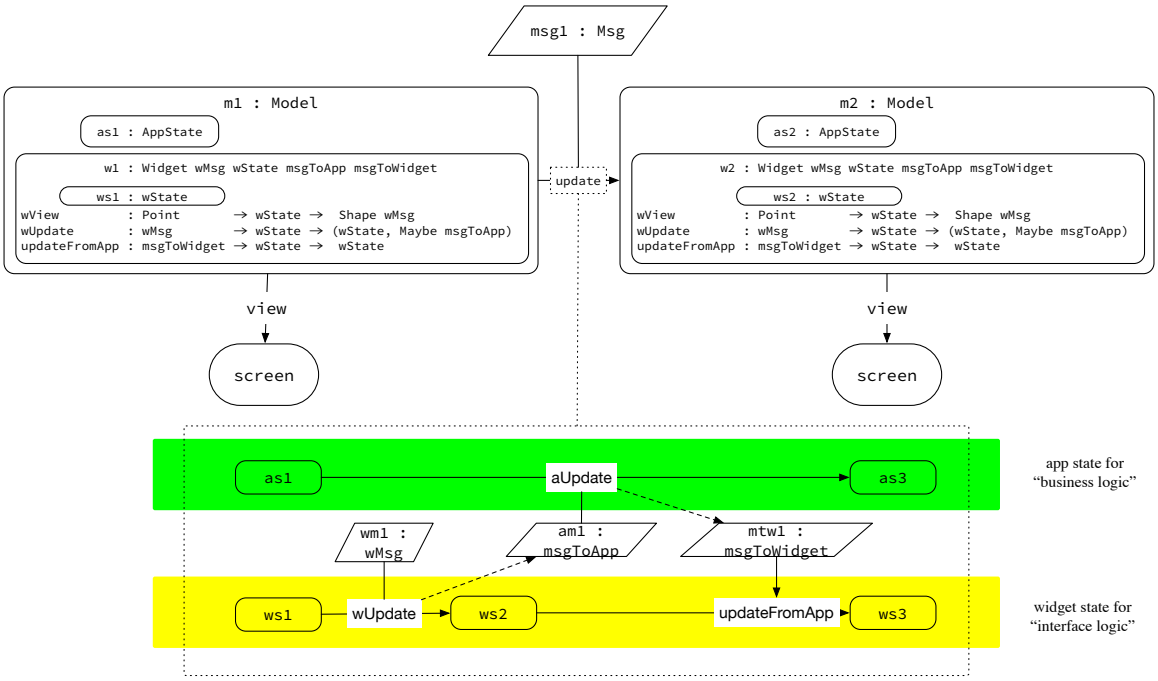

Figure 6.3: Exploded view of the `update` function from Figure 3.4 in an app using our widget library.

This record seems to break the separation of model, view and update, since it contains all three aspects of the widget tree. This is necessary for a dynamic user interface, because at run-time, the location of subwidgets could change, requiring the

view functions to be updated. Whether the interface should change dynamically is a design decision which will depend on the context. Norman's principle of consistency demands that widgets be found in consistent places, but in an "editor" app capable of editing different types of data, it may be necessary to present different controls dependent on the actual data loaded.

Figure 6.3 shows how the model-view-update (top) separation is partially maintained, in the detailed view. In the middle, details of the state show that the model contains the root widget, `w1`, which contains a state, but also its view and two update functions. The `Widget` type constructor takes several type components, including: Widget Message (`wMsg`): Messages generated by widgets in response to user interactions, such as clicks or drags. Widget State (`wState`): The internal state of the widget, which can change based on user actions and messages received. Message to App (`msgToApp`): Messages from widgets to the main application logic (business logic), indicating changes or events. Message to Widget (`msgToWidget`): Messages from the main application logic to widgets, instructing them to update or change state.

At the bottom, we show how the two state components influence each other. Similar to the overall pattern, changes in state only happen in response to messages, and all state changes are localized in the update functions. Starting on the left, we see a message `wm1:wMsg` encoding a user action such as a button click is received by the run-time system, wrapped in a type wrapper for widget messages. The main update function unwraps it and calls the widget update function, `wUpdate`, with the extracted message. When we examine the type for `wUpdate`, we notice that it resembles the type for `update`, except for having a `Maybe msgToApp`. This implies a similarity to the "command," which enables updates of an Elm app to generate asynchronous

functions or messages that the runtime system interprets as external function calls or actions, like sending an `http` request. If an `am1:msgToApp` message is generated, the top-level update function will call the `aUpdate` function defined at the application level to handle app/business-logic state transitions triggered by user interactions within the widget. This update again as a similar form, and can optionally generate a message to update the widget. The top-level update now calls `updateFromApp` with the Message to Widget(`mtw1`) and the Widget State (`ws1`) as input and updates the Widget State (`ws2`) based on the received message. Thus the widget state is updated in two stages. For example, an action which requires sychronization with a database or peer client apps could initially change widget state to indicate that the request has been received, following Norman's principle of Feedback, signal to the business logic that the action should be performed, and upon receiving the `mtw1` change the widget to indicate success or failure of the action.

Finally, the `Widget` also contains a `wView`. This function takes a Point and a `wState` as input and produces a Shape with a corresponding Widget MessagewMsg. The `wView` function is responsible for rendering the widget's visual representation based on the model's current state and the user's interactions.

One crucial design principle discussed was the separation of concerns between the business logic and interface logic. The "app state" (`as1`) is the overall state of the entire application, representing the business logic. It's not a type variable because it likely refers to a specific, well-defined data structure. "widget state" (`ws1`) is a type variable that represents the state of a single widget. It is part of the Widget type and allows different widget types to have distinct state representations. The use of a type variable allows flexibility in defining the specific state representation for each

widget type.

The main way for the "app state" (`as1`) and "widget state" (`ws1`) to communicate is through messages. Messages flow from widgets to the main application logic `msgToApp` and from the main application logic to widgets `msgToWidget`. These messages trigger updates and state changes in both components. "app state" and `wState` serve different purposes and have different levels of generality. The "app state" captures the application's high-level state, while "widget state" deals with the specific state of individual widgets within the application.

We draw a line between "app state" and "widget state" to represent the division. The states are stored immutably and are updated through update functions. Communication between the two components occurs through two types of messages: `msgToApp` and `msgToWidget`.

For example, we have a toggle with a view function. When the toggle is clicked, a message is generated and received by the main update function. The message is wrapped in the widget type and sent to the widget update function for the widget tree. The widget update function modifies the widget state and may produce a message of its own. This new message is then interpreted by the app update function to modify the app state, potentially producing another message for the widget. The process continues as needed.

For simple interactions, this flow is seamless and not noticeable to the user. However, if the interaction involves synchronization with a server or other client applications, the widget's state might be changed by the widget update function to indicate that the click was received, but the visual indication may not confirm the acceptance of the action yet. The update function in such cases may cause a message to be sent to

the server or other clients for synchronization. Only when the state is synchronized, the message to the widget will be generated, allowing the `updateFromApp` function to change the state of the widget to indicate the updated live state.

In a distributed application scenario, multiple users may be interacting with the same system simultaneously. For example, when typing, a spinning beach ball might appear to indicate ongoing changes. Once the text is synchronized with the server, the spinning beach ball disappears, and the updated text is displayed. Meanwhile, other users may also be typing, leading to additional text updates.

6.1.2 Defining the `Widget` type

The `Widget` type alias provides a structured way to define and manipulate widgets in the application, encapsulating their properties, rendering behaviour, and event-handling logic. The type alias has multiple type parameters (`wMsg, wState, msgToApp, msgToWidget`), allowing flexibility in the types of messages and models that a widget can work with.

Listing 6.1: Widget's Type definition

```
type alias Widget wMsg wState msgToApp msgToWidget =
    { width        : Float
    , height       : Float
    , pos          : (Float, Float)
    -- Function to transform the Shape (eg, add an outline)
    , outline      : Maybe (Shape wMsg -> Shape wMsg)
    -- Inital model value
    , model        : wState
```

```
    -- Function to render the widget
    , wView        : Point      -> wState -> Shape wMsg
    , wUpdate      : wMsg       -> wState -> (wState, Maybe msgToApp)
    , updateFromApp : msgToWidget -> wState -> wState
    }
```

Recall that Elm records are similar to structures in C and can be used to group related data together, making it easier to pass and manipulate that information as a whole. One of the advantages of using records is that they allow us to organize and encapsulate related instance properties and methods. If this were an OO framework, these would be properties and methods of an abstract class. However, in Elm, they are simply fields. Instead of inheritance, we use parametric polymorphism to differentiate the different types of widgets, with a stronger level of type safety, and potentially lower overhead since all information needed for code specialization is encoded in compile-time type variables. For example, in our case the `wView` field inside our type alias is parametrized by `wState` and `wMsg`—remember that lower-case initial letters indicate type *variables*. By using a record, we can keep all the relevant information together, making it convenient to work with during the GUI construction. They provide a convenient and structured approach to handling data, enhancing code readability and maintainability. This is similar to the role of objects in OO programming, but unlike objects, our use of records does not add additional overhead once the GUI is constructed.

Within the presented code snippet, a collection of variables and functions is utilized to establish and manipulate a widget. This section endeavors to provide hands-on view of these elements, shedding light on their composition and functionality.

The `width` and `height` variables represent the dimensions of the widget. The `outline` function is optional and can be used to apply an outline to the widget by taking a `Shape` and message as input and returning a modified shape [40]. The `pos` variable is a tuple that represents the position of the widget. The `model` field represents the data associated with the widget and is initialized as required to render and interact with the view. The `wView` function takes a `Point`, and a model as input, and returns a `Shape` message. A `Point` type is represented as a tuple with two elements used to specify the position of the rendered shape within the two-dimensional space of the parent widget. By passing different `Point` values, you can control the position of the view in relation to other elements on the screen. This function defines how the view should be rendered based on the provided information. The `wUpdate` and `updateFromApp` functions update the widget state, as described above.

The purpose of `msgToApp` is defined by the high-level code. The Widget is a component that represents a user interface element, such as a button or a toggle switch. It is responsible for processing user events, such as clicks, and generating low-level events or messages based on those interactions. For example, in the code with nested side-by-side widgets, the widget generates messages of type

```
SideBySideView.Msg (SideBySideView.Msg (ToggleView.Msg) (LabelView.Msg))
```

or

```
SideBySideView.Msg ButtonView.Msg ButtonView.Msg
```

based on the user interactions.

The `msgToApp` concept helps to separate the low-level event processing within the widget from the higher-level business logic of the application. The widget is focused on

processing events and making low-level changes to its state or behavior. It shouldn't directly handle complex business logic or perform high-level actions. Actions are defined by the widgets, and the high-level code can assign `msgToApp` messages to them or `Nothing`.

To perform higher-level business logic or trigger actions, the programmer defines custom messages that encapsulate the necessary information or instructions for the business logic to handle.

The application programmer is responsible for defining the types `msgToApp` (with constructors for each message the business logic requires) and `msgToWidget` (whose nested type mirrors the nesting of the widgets). When the widget is created, one or more actions can be set up to send these custom messages to the business logic. This allows the widget to be connected to but independent of the business logic.

6.1.3 Building a `ToggleView`

The `build` function in the given context can be compared to an initializer or constructor in object-oriented programming. In our case, the build function doesn't directly create objects, the build function is responsible for constructing a record that serves as a data structure to organize and manage the necessary information for building the view hierarchy. It is particularly helpful while constructing composite views, which are views composed of multiple smaller views. The build function takes input parameters, such as the dimensions and properties of the sub-views, and uses them to assemble the record that represents the composite view. The record is not retained once the view hierarchy is built. It serves as a temporary data structure during the construction process unlike an object in object-oriented programming approaches.

Now let's see the type annotation of our build function,

```
build : Float -> String -> Actions msgToApp
    -> Widget ToggleView.Types.Msg State msgToApp ToTree
```

The function takes a `Float` (representing the width of the toggle view), a `String` (representing the label text of the toggle view), and an `Actions` value specific to the toggle view. It returns a view of type (`Widget ToggleView.Types.Msg State msgToApp ToTree`), which is a specialized type based on the `Widget` type defined in the `Types` module.

In the `ToggleView.Types` module we define alias `Msg` which represents the type of messages that the toggle view can receive. In this case, it is a `Bool`, indicating that the toggle view can receive messages that represent Boolean values.

We define an alias `State` which represents the internal state of the toggle view. It has Boolean fields `on` and `active`, and a string field `label`.

We have an *abstract data type* `ToTree` which represents messages that can be transformed into a tree structure. It includes constructors `SetLabel, SetOn, SetActive`, each with its own associated data types. These constructors are used to wrap the `Msg` type and provide a structured representation of the messages that can be sent to the toggle view[1].

[1]When we want to create our own custom types in Elm we use an *Algebraic Data Type (ADT)* For instance, in
`type ToTree = SetLabel (Shape Never)`
the `SetLabel` is called a *data constructor*. This is because they can be considered as a constructor with parameters. But constructors can construct values without associated data. In
`type msgToApp = MoreToggles | LessToggles`
the `MoreToggles, LessToggles` are values by themselves. They are called called a *nullary data constructors*, that is, a constructor that takes no arguments. The `ToTree` and `msgToApp` types are both ADTs. Custom types used to be referred to as "union types" in Elm. Names from other communities include tagged unions and ADTs. [95, 33].

Finally, we have an alias `Actions` which represents the actions that can be performed on the toggle view. It has fields `switchOn, switchOff`, both of type `Maybe`[2] `msgToApp`. These fields allow you to specify optional actions to be performed (by sending messages from the tree to the application code) when the toggle view is switched on or off. The `Maybe` type indicates that the actions can be `Just` a message or `Nothing` if no action is needed.

Now back to the `build` function, it takes arguments for the width, label text, and actions associated with the toggle view. Inside the `build` function, it uses the types defined in `ToggleView.Types` that we have seen above to configure and initialize the toggle view.

Listing 6.2: Example of build function

```
build : Float -> String -> Actions msgToApp
    -> Widget ToggleView.Types.Msg State msgToApp ToTree
build w txt actions =
  { width = w
  , height = buttonHeight
  , pos = (0,0)
  , outline = Nothing
  , model = { on = False, active = True, label = txt }
  -- draws the toggle view
  , wView = \ _ model
```

[2] `type Maybe a = Just a | Nothing` represents values that may or may not exist. It can be useful if you have a record field that is only filled in sometimes. `Maybe` is a core type in Elm that allows you to model the idea of optional values. Sometimes, we are not sure whether a value is returned. To create a value of type `Maybe`, we could either use the `Just` data constructor or the `Nothing` constant [95].

```
            -> [ roundedRect (w-0.25) (buttonHeight-0.25)
                (0.5*(buttonHeight - 0.25))
                   |> filled activeClr
                   |> addOutline (solid 0.25)
               , text model.label |> fixedwidth |> size 6 |> filled black
                   |> move (-0.5*w + buttonHeight, -0.125*buttonHeight)
               , circle (0.3*buttonHeight)
                   |> filled ( if model.on then rgb 0 0 255 else
                      activeClr )
                   |> addOutline (solid 0.25) black
                   |> move (-0.5*w + 0.5*buttonHeight, 0)
               ]
                 |> group
                 |> move (0.5*w, 0.5*buttonHeight)
                 |> notifyTap (not model.on)

, wUpdate = \ msg model -> ( { model | on = msg }
                   , case msg of
                          True -> actions.switchOn
                          False -> actions.switchOff
                   )
, updateFromApp = \ tmsg model -> case tmsg of
                          SetOn isOn -> ( { model | on = isOn },
                             Nothing)
                          SetLabel label -> ( { model | label = label
                             }, Nothing)
```

```
                    SetActive isActive -> ( { model | active =
                        isActive }, Nothing)
  }
```

The `wView` function within the build defines the appearance of the toggle view. It creates a list of graphical elements using SVG functions provided by the `GraphicSVG` module. These elements include a rounded rectangle, text, and a circle. The appearance of these elements depends on the state of the toggle view model.

The `wUpdate` function specifies how the toggle view model should be updated based on the received messages. It also defines the corresponding actions to be performed based on the updated model.

The `updateFromApp` function specifies how the toggle view model should be transformed into the tree structure based on the received messages.

```
updateFromApp : msgToWidget -> wState -> wState
```

This function takes two arguments: `msgToWidget`, representing the message to be received, and `msgToWidget`, representing the widget state to be transformed. The function then performs pattern matching[1] on the `msgToWidget` argument to determine the specific transformation to apply. If the `msgToWidget` belongs to the alternative `SetOn`, it updates the `on` field of the widget state to the provided `isOn` value. The updated widget state is returned along with `Nothing`, indicating that no additional message needs to be sent, similarly for `SetLabel, SetActive`. The `updateFromApp`

[1]Pattern matching is the act of checking one or more inputs against a pre-defined pattern and seeing if they match. In Elm, there's only a fixed set of patterns we can match against, so pattern matching can be checked by the compiler. The `case` expression works by matching an expression to a pattern. When a match is found, it evaluates the expression to the right of `->` and returns whatever value is produced [139].

function performs the appropriate updates on the toggle view's state and returns the updated state.

Listing 6.3: Creating a toggle view

```
type alias MsgToApp = ()

type alias AppState = ()

type alias MsgToWidget = ToggleView.ToTree


notToggleAction = {switchOn = Nothing , switchOff = Nothing}


type alias Model = { ...
                   , appState : AppState
                   , theView : Widget (ToggleView.Msg) (ToggleView.State)
                   MsgToApp
                   MsgToWidget
                   }


init : Model
init = { ...
       , appState = ()
       , theView = ToggleView.build 40 "Blue" notToggleAction |>
           moveView(90,30)
       }


view: Model -> Browser.Document Msg
view model =
```

```
  { body = [createCollage collageWidth collageHeight <|
              [drawIn 192 128 model.theView Nothing
                 |> GraphicSVG.map Widget
              ]
            ]
  , title = appTitle }


type Msg = Tick Float
         | ....
         | Widget (ToggleView.Msg)
         | SendToTree MsgToWidget


update: Msg -> Model -> (Model, Cmd Msg)
update msg model =
 let
    ...
 in
  case msg of
    ...
    Widget wMsg ->
        let
          thisView = model.theView
          newModel = thisView.update wMsg thisView.model
          newAppState = model.appState
          newView = { thisView | model = newModel }
        in
```

```
        ({ model | theView = newView , appState = newAppState }, Cmd.none)

    SendToTree msgToWidget ->
        let
            thisView = model.theView
            newModel = thisView.updateFromApp msgToWidget thisView.model
            newView = { thisView | model = newModel}
        in
            ({ model | theView = newView }, Cmd.none)
```

We have a type alias for the `Model`, which represents the overall model for your application. It contains various fields such as `appState`, and `theView`. The `theView` field is of type `View` and represents the view component of our application. We initialize our model with some initial values.

The `theView` field is created by calling `ToggleView.build` with the width of the toggle view, label, and action to be performed, basically to update the state based on the action message. The result is then passed to `moveView (90,30)` to adjust the position of the view.

The Msg[1] type includes message constructors such as `Widget, SendToTree`, etc. These constructors represent different types of events that can occur in the application, like interactions with the ToggleView component, and more. When a message is received in the `update` function, it is pattern matched to determine how to handle that specific message. Based on the type of message received, appropriate actions can

[1]Messages (represented by the `Msg` type) serve as a way for different parts of the application to communicate and trigger updates according to the Elm Architecture. Messages are used to represent user interactions, events, or actions that occur within the application.

be taken, such as updating the model, triggering side effects, or sending messages to other components. In our example, `type alias msgToApp = ()` This defines an alias `msgToApp` for the unit type[1] `()`. It indicates that the `msgToApp` type carries no information, because in this example there is no business logic yet. Similar to `msgToApp`, `type alias AppState = ()`, this defines an alias `AppState` indicates that there is no information in `AppState`. This would never happen in a real app.

The definition `notToggleAction = {switchOn = Nothing, switchOff = Nothing}` creates a record `notToggleAction` with fields `switchOn` and `switchOff`, based on the action the turning switch is set to On or Off.

The update function handles different message variants and updates the model accordingly:

- The `Widget` message constructor is used to handle messages related to a specific widget in the application. It wraps a message of type `ToggleView.Msg` in this case.
- When a `Widget` message is received, `update` function extracts the `ToggleView.Msg` and passes it to the `update` function of the topView (the root widget) using the `thisView.update` function call.
- The `update` function of the app returns a new model and a set of commands.
- The `SendToTree` message constructor is used to send a message to the widget tree (`ToggleView`) directly, without going through a specific widget.

[1]Tuples are types but they are dependent on their length as well as the types of their components, so there is theoretically an infinite number of tuple types. The empty tuple () is also a type which can only have a single value: (). This value is read as "unit" and is the common way to denote an empty value with no specific meaning.

- When a `SendToTree` message is received, the `update` function extracts the `msgToWidget` message and passes it to the `updateFromApp` function of the topView using the `thisView.updateFromApp` function call.
- The `updateFromApp` function of the widget tree processes the message and returns a new model, which are used to update the application state and the widget tree accordingly.

These message constructors allow communication with specific widgets or the widget tree as a whole, enabling updates and interactions within the application's view hierarchy.

In the view function, with the `drawIn` function that prepares the necessary parameters and transformations to render a view based on its position, dimensions, and model. This applies clipping and positioning operations to ensure the view is displayed correctly on the canvas or screen. We use `GraphicSVG.map Widget` to wrap messages of the top-level widget type with the constructor `Widget` so they have the application-level `Msg` type. Functions called `map` are commonly used to transform the contents of a container while preserving the container structure. We can think of this as preserving the innermost message contents, but wrapping it in types to make the application type safe. This introduces overhead similar to the overhead of object composition in OO languages, but because it is all determined statically, it might be possible for a compiler to optimize it away. Although it does introduce overhead, the use of the higher-order map function keeps the notation compact.

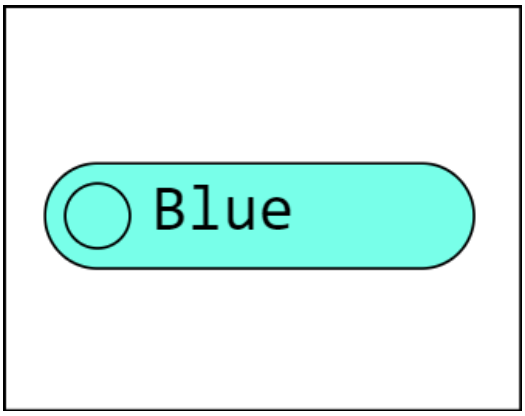

Figure 6.4: ToogleView widget by default

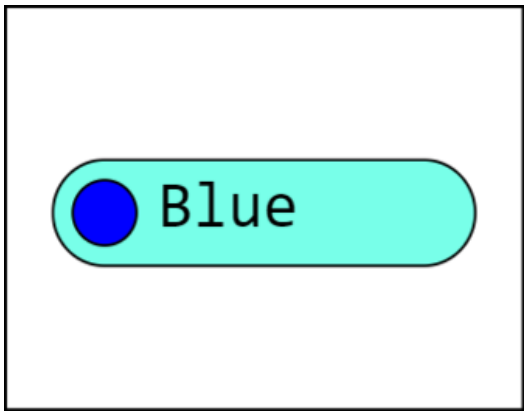

Figure 6.5: ToogleView widget when selected

6.2 Building a `SideBySideView`

A `SideBySide` view is a widget that can be used to create a composite view that is an aggregate of multiple subviews horizontally.

We use the `build` function to create this widget, the build function takes the gap between the two subviews and the subviews themselves as arguments. It constructs and returns a `Widget` with the combined `SideBySideView`. The `Widget` is discussed in (6.1.2). To the view function we are passing the `width` and `height` to specify the dimensions. The `SideBySideView.outline` is an optional outline around the view, and `pos` represents the position of the view. The app-level `model` contains the state of the `SideBySideView`, which includes the states of the two subviews.

The `wView` is a function that renders the view based on the current state, while `wUpdate` handles messages and updates the state accordingly. It delegates the messages to the corresponding subview based on whether the message is (wrapped in) `Left` or `Right`. Here `Left` or `Right` means either the subview on the left or right.

The `updateFromApp` handles messages sent to the subviews' update functions and

returns the updated state and any resulting tree command. The function takes two arguments: `msgToWidget` and `model`. The `msgToWidget` is a value of `type ToTree toA toB`. It represents the message to be sent to one of the subviews. It can be either `ToA subMsg` or `ToB subMsg`, indicating which subview should receive the message. The `model` is a tuple representing the current state of the `SideBySideView`. It contains the `states` of the two subviews, `(subModelA, subModelB)`.

If `msgToWidget` is `ToA subMsg`, this means that the message should be sent to `subA`. The function delegates the update to subA's `updateFromApp` function by passing `subMsg` and the first element of the model tuple `(Tuple.first model)`. It captures the updated state `newA` and any resulting tree command `msgToApp`. Similarly for the case that `msgToWidget` is `ToB subMsg`. Based on which subview received the message, the function returns a new tuple with the updated state of the corresponding subview and the unchanged state of the other subview, along with any resulting tree command.

Listing 6.4: Creating a SideBySideView

```
type alias Model = {
   ...
   ...
   ,theView : View (SideBySideView.Msg ToggleView.Msg LabelView.Msg)
                (SideBySideView.State ToggleView.State LabelView.State)
   ...
       }
init : Model
init = {
   ...
   ...
```

```
   theView = ( SideBySideView.build
                   20
                   (ToggleView.build 40 "Blue" notToggleAction)
                   (LabelView.build 40 displayText)
             ) |>moveView (50,50)
       }
type Msg = Tick Float
        ...
        ...
        | Widget (SideBySideView.Msg (ToggleView.Msg)(LabelView.Msg))

displayText = text "Elm"
           |> filled darkGreen
           |> scale 0.5
```

The code would be similar to the previous example 6.3, except we need to alter the view with the `SideBySideView`. The `SideBySideView` is given two subviews, `ToggleView` and `LabelView`. `LabelView` is a widget that can display any `Shape` and clip it to the allowed rectangle. In this case, we display text.

LabelView `build` is a function that constructs a `Widget` for the label widget. It takes a Float value for the width, a `Shape Never`[1] representing the visual shape of the label (for example, text), and returns a `LabelView`. We construct the widgets using `SideBySideView.build`, `ToggleView.build`, and `LabelView.build` functions.

The `Model` record type alias encapsulates the different fields that store the state

[1]The `Shape` produces messages of type `Never`, which has zero cardinality and therefore has no values. In turn, this means that the shape cannot have any event listener/message producers associated with it

of the application, including the `theView` field that represents the current view in the GUI and its associated messages and state.

The first parameterized type `(SideBySideView.Msg ToggleView.Msg LabelView.Msg)` represents the type of messages that can be sent to the view. It combines the message types specific to the subviews `ToggleView`, and `LabelView` using the type combinator `SideBySideView`. The second parameterized type (`SideBySideView.State ToggleView.State LabelView.State`) represents the state of the view. It combines the state types specific to the subviews `ToggleView`, and `LabelView` using the type combinator `SideBySideView`. Each of these views defines its own set of messages (`ToggleView.ToTree` and `LabelView.ToTree`) that can be sent to the corresponding view's internal tree structure. By combining these different message types with `SideBySideView.ToTree`, the `msgToWidget` type alias represents a message that can be sent by the main app logic to the view hierarchy. It incorporates the specific messages for each view within it. This strong typing ensures type safety. It is impossible to send an message the the wrong type of subwidget. This comes at the expense of requiring the messages to be wrapped in constructors corresponding to the path from the root of the view hierarchy to the target widget.

`Widget (SideBySideView.Msg (ToggleView.Msg)(LabelView.Msg))` is the type of a message generated by a widget in the view hierarchy. It combines the messages specific to the subcomponents, following the familiar pattern. This allows the application to handle messages from different widgets in a unified way. This serves as a way to propagate messages received from the operating system by the top-level `update` function, where they can be recursively piped to the appropriate subwidget by pattern matching on the different variants of messages in the update function, allowing for

modular and extensible message-handling logic.

Figure 6.6: SideBySideView widget with Toggle and LabelView

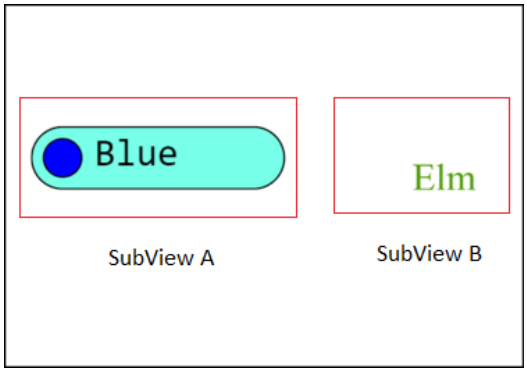

Figure 6.7: Highlighting distinct subviews with added red boxes

6.3 Building a nested `SideBySideView` with `Toggle`, `Label`, and `ButtonView`

The `theView` is constructed using various view builders (`ToggleView.build`, `LabelView.build`, `ButtonView.build`) and organized using `SideBySideView.build`, which creates a tree-like structure. It follows the same basic pattern as the previous examples.

This example introduces state at the app level, albeit very simple state. It shows how messages to and from the widget tree can be used, with the example of getting messages when buttons are clicked, updating app state and reflecting that change in a label. To facilitate communication between the tree structure and the application, the code defines an abstract data type called `msgToApp`, which has two possible values: `Increase` and `Decrease`. The `AppState` alias represents the application state, which in this case is a simple integer.

The `updateFromTree` function is a helper function that updates the application

state (`AppState`) based on the message received from the tree structure (`msgToApp`). It uses pattern matching to handle different cases: if the message is `Just Increase`, the `AppState` integer value is incremented by 1; if the message is `Just Decrease`, the `AppState` integer value is decremented by 1. If the message is anything else (_), the `AppState` remains unchanged. The updated `AppState` is returned.

The `update` function handles different message types using pattern matching. If a `Widget wMsg` message is received, it means a widget-related message is being processed. The function retrieves the current `theView` from the model and updates it using the corresponding update function for that widget. It then updates the application state based on the received tree commands using the `updateFromTree` function. Using the `SendToTree` message is being sent to the tree structure the corresponding `updateFromApp` function is used to update the model. The view is updated with the new model.

Listing 6.5: Creating a nested SideBySide View with Toggle, Label, and Button Views

```
type MsgToApp = Increase | Decrease

type alias AppState = Int

type alias MsgToWidget = SideBySideView.ToTree
                (
                SideBySideView.ToTree
                    (ToggleView.ToTree)
                    (LabelView.ToTree)
                )
                (SideBySideView.ToTree ButtonView.ToTree ButtonView.ToTree)
setLabel : String -> MsgToWidget
```

```
setLabel str =
    SideBySideView.ToA <|
        SideBySideView.ToB <|
            LabelView.SetLabel (text str |> filled black |> scale 0.5)

notToggleAction = {switchOn = Nothing , switchOff = Nothing}

type alias Model = {  ...
           , theView : View
             (SideBySideView.Msg (SideBySideView.Msg (ToggleView.Msg)
                 (LabelView.Msg)) ButtonView.Msg ButtonView.Msg))
             (SideBySideView.State (SideBySideView.State
                 (ToggleView.State) (LabelView.State))(SideBySideView.State
                 ButtonView.State ButtonView.State))
             MsgToApp
             MsgToWidget
             ...
           }

initalDisplayText = text "1"
           |> filled black
           |> scale 0.5

init : Model
init = { ...
       , theView = SideBySideView.build 2
```

```
            (   SideBySideView.build 20
                ((ToggleView.build 40 "Blue" notToggleAction))
                ( LabelView.build 30 initalDisplayText)
            )
            (SideBySideView.build 5
                (ButtonView.build 20 "+" {click=Just Increase})
                    (ButtonView.build 20 "-" {click=Just Decrease})
            ) |> moveView (0,50)
      }
type Msg = ...
        | Widget (SideBySideView.Msg (SideBySideView.Msg
                                    (ToggleView.Msg) (LabelView.Msg))
           (SideBySideView.Msg ButtonView.Msg ButtonView.Msg))
        | SendToTree MsgToWidget


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
 let
    ...
 in
  case msg of
    ...
    Widget wMsg ->
        let
          thisView = model.theView
          newModel = thisView.update wMsg thisView.model
```

```
          newAppState = updateFromTree model.AppState
          newView = { thisView | model = newModel}
        in
          ({ model | theView = newView , AppState = newAppState }, newMsg
              <| SendToTree <| setLabel <| String.fromInt newAppState )

    SendToTree MsgToWidget ->
        let
            thisView = model.theView
            newModel = thisView.updateFromApp MsgToWidget thisView.model
            newView = { thisView | model = newModel}
        in
            ({ model | theView = newView }, Cmd.none)

updateFromTree : Maybe MsgToApp -> AppState -> AppState
updateFromTree MsgToApp MppState =
  AppState + case MsgToApp of
                Just Decrease -> -1
                Just Increase -> 1
                _ -> 0

newMsg : msg -> Cmd msg
newMsg = Task.perform identity << Task.succeed
```

The provided figures, Figure 6.8 and Figure 6.9 give visual representations of key components within the application's user interface. Figure 6.8 showcases the

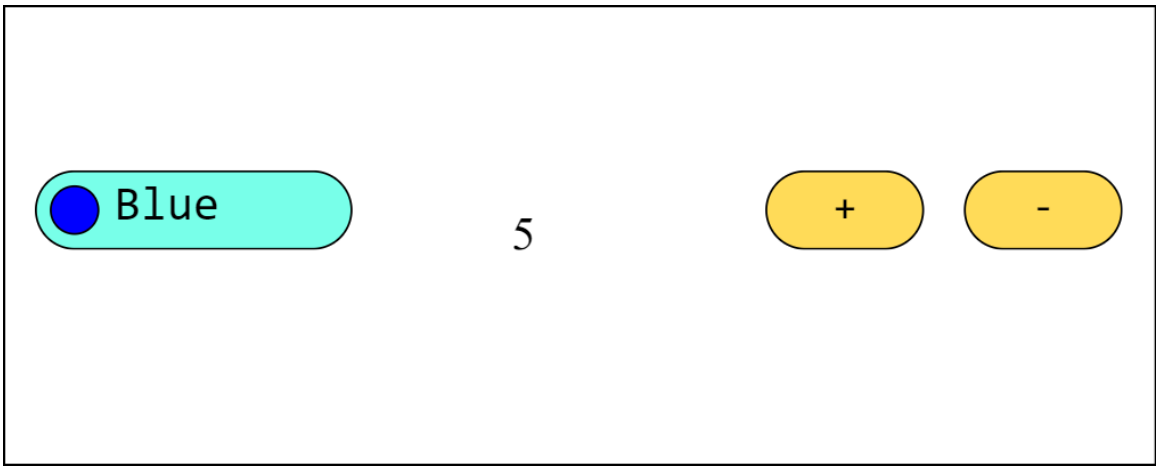

Figure 6.8: Nested SideBySideView with Toggle, LabelView and ButtonViews

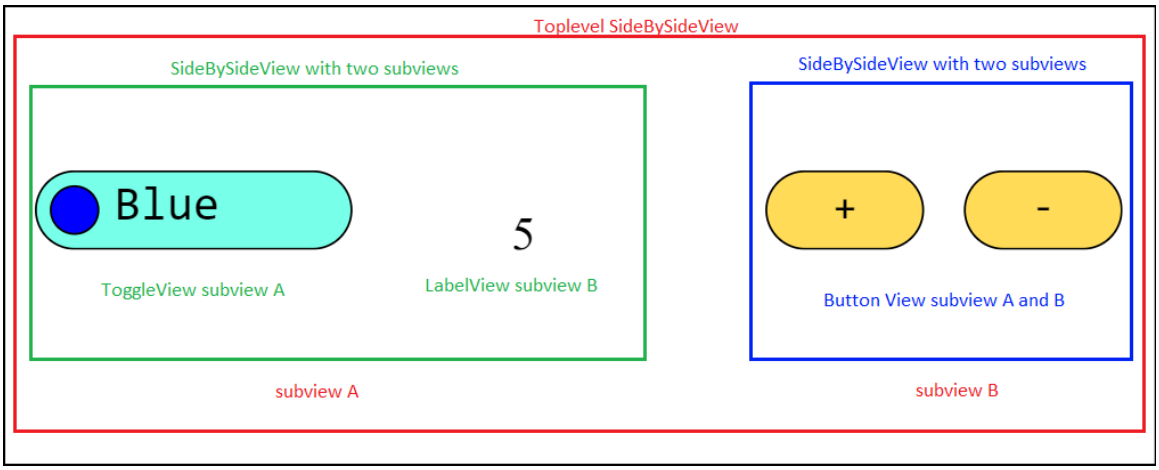

Figure 6.9: Highlighting distinct nested subviews with added color boxes

`SideBySideView`, featuring the inclusion of the `Toggle`, `LabelView`, and `ButtonView` elements. This depiction allows for an understanding of the layout and arrangement of these specific subviews within the top-level `SideBySideView`. In Figure 6.9, the emphasis is on highlighting distinct nested subviews, accentuated by the addition of green and blue boxes that represents the subviews. This aids in illustrating the hierarchical structure and relationship between the nested subviews, offering insights into the composition.

Chapter 7

Conclusion

We have answered all the research questions:

RQ1 *Is it practical to build a GUI Toolkit using purely immutable data (internally)?* Based on the work reported in Chapter 6, it has been established that building a toolkit using purely immutable data is indeed practical. It is worth noting that strong typing plays a crucial role in identifying type-related errors early in the development process. In Elm, almost all run-time errors including all typing errors are eliminated. Strong typing encourages developers to provide explicit type annotations, resulting in a codebase that is easier to understand, maintain, and debug but in our current code, including explicit types increases the amount of boilerplate code we must have.

RQ2 *What are the advantages of Declarative UIs, from both academic and profession points of view?* From an *academic perspective*, there are several advantages to using declarative approaches. Academics have long favoured declarative programming because it allows developers to work at a higher level of abstraction,

which can reduce the number of errors caused by low-level implementation details. In the *professional realm*, there has been a growing recognition of this benefits offered by declarative UIs. Professionals are increasingly aligning with the views of academics in this regard. The adoption of declarative UIs has been driven by the realization that traditional UI programming approaches can be time-consuming, expensive, and yield limited value.

RQ3 *Are Declarative UIs the future?* It is worth noting that while declarative programming has been embraced in certain domains, such as database programming with SQL, it has not been as prevalent in UI programming until recently. A shift is evident in trends observed on platforms like Stack Overflow, where discussions and questions related to declarative UI frameworks have seen significant growth and engagement as seen in section 4.1.1. Furthermore, major vendors in the software industry are now actively promoting the adoption of declarative UIs, emphasizing the productivity gains and improved user experiences that can be achieved. Given that vendors have committed to the transition, and developers are showing strong interest, it seems very likely that declarative UI is the future of GUI development.

7.1 Future Work

Chapter 6 establishes that most aspects of a practical GUI toolkit can be implemented using purely immutable data, but one feature which still needs to be demonstrated is keyboard, focus-based navigation. This is often omitted in UI experiments, and is often a confusing aspect of vendor-supplied toolkits, e.g., requiring the concept of

"delegate" objects in UIKit. It is, however, required by users with visual impairments, and much appreciated by "power users" who can more efficiently perform many tasks using their keyboard than with a mouse.

But even in the simple examples contained in Chapter 6, it is clear that constructing the types for messages, state, etc., would be unacceptably burdensome to many front-end developers. Fortunately, these types can be constructed from the view hierarchy, and we have tested this by adding a type construction function to the `View` record. UI developers are used to working with drag-and-drop GUI builders, and we propose, in the near future, to develop such a builder which would generate all the required types as well as the calls to the `build` functions.

In order to investigate the current and potential knowledge transfer from academic discussions of immutable data to the developer community, we propose adopting a structured research approach. In this formative study, we did find evidence that declarative UIs are being rapidly, and non-linearly adopted, but since we didn't start with a hypothesis and a methodology for examining sources, we cannot make any conclusions. In the future, hypotheses should be developed, and systematic cataloging of the numerous forums and blog posts, and contributions to open-source software could support or refute these hypotheses. Additionally, developers should be surveyed about their sources of information, decision processes behind architectural decisions, and differences they have seen in development practices over time, particularly in the context of framework adoption. This would provide an opportunity to assess the perceived importance of specific features, including immutable data types, declarative approaches, and interface design, within the framework selection process. To enhance

the reliability and validity of the findings, it is essential to design the survey in a manner that encompasses a diverse range of developers, considering various programming domains, industries, and experience levels. Alternatively, the research can focus on a specific subset of developers initially, while acknowledging that the findings may not fully represent the entire developer population. By utilizing well-crafted questions and employing appropriate sampling strategies, the collected data can be analyzed using statistical techniques to quantify the significance attributed to the aforementioned features by developers. Additionally, measuring the prevalence of discussions surrounding these concepts within the developer community can offer insights into their practical implications and real-world adoption.

Another direction of research would be to evaluate the use of this framework as a teaching tool for students just learning design. For example, at McMaster, students are introduced to design in the first-year course "Introduction to Software Design using Web Programming (CS 1XD3)", and interviews could be used to determine the depth of understanding which results from using a purely immutable framework versus a Declarative UI framework versus a more conventional framework.

Appendix A

Android Code Examples

A.1 Architectural patterns in action

Let's see a few android examples implementing the concepts explained in the Chapter 3, this is an example where the user wants to add two numbers and is presented with a view that has two editable fields, a text view to view the result, and a button to perform an action after entering the numbers.

A.1.1 Model - View - Controller (MVC)

The below code shows the implementation of MVC,

```
public class MainActivity extends AppCompatActivity {
    private EditText mNumberOneEditText;
    private EditText mNumberTwoEditText;
    private TextView mResultTextView;
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    /* Setting the view */
    setContentView(R.layout.activity_main);
    mNumberOneEditText = findViewById(R.id.number_one_edit_text);
    mNumberTwoEditText = findViewById(R.id.number_two_edit_text);
    mResultTextView = findViewById(R.id.result_text_view);

    Button addButton = findViewById(R.id.add_button);
    /* Controller Logic */
    addButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            int numberOne =
                Integer.parseInt(mNumberOneEditText.getText().toString());
            int numberTwo =
                Integer.parseInt(mNumberTwoEditText.getText().toString());

            CalculatorModel calculatorModel = new
                CalculatorModel(numberOne, numberTwo);
            int result = calculatorModel.addNumbers();

            mResultTextView.setText(String.valueOf(result));
        }
```

```
        });
    }
}

 /* Business Logic / Application State */
class CalculatorModel {
    private int mNumberOne;
    private int mNumberTwo;

    public CalculatorModel(int numberOne, int numberTwo) {
        mNumberOne = numberOne;
        mNumberTwo = numberTwo;
    }

    public int addNumbers() {
        return mNumberOne + mNumberTwo;
    }
}
```

In this example, the *MainActivity* class represents the View, responsible for handling user interactions and displaying the results. The *CalculatorModel* class represents the Model, and encapsulates the data and logic for the application. The *Controller logic* is implemented by the button click listener in the MainActivity class, which takes the user input, updates the Model with the new data, and updates the View with the result. Checkout the full code here [11] : MVC Example

A.1.2 Model - View - Presenter (MVP/ Passive View)

```
/* MainActivity would be quite as same as MVC but with the below
    changes.It presents the view, also you can notice we have implemented
    the MainView */
public class MainActivity extends AppCompatActivity implements MainView {
        mMainPresenter = new MainPresenter(this, new CalculatorModel());
        //Inside the onclick listener
        mMainPresenter.addNumbers(numberOne, numberTwo);
    }

    //Updates the view with the current state
    @Override
    public void updateResult(int result) {
        mResultTextView.setText(String.valueOf(result));
    }
}

/* Presenter logic*/
class MainPresenter {
    private MainView mMainView;
    private CalculatorModel mCalculatorModel;

    public MainPresenter(MainView mainView, CalculatorModel
        calculatorModel) {
        mMainView = mainView;
```

```
        mCalculatorModel = calculatorModel;
    }

    public void addNumbers(int numberOne, int numberTwo) {
        int result = mCalculatorModel.addNumbers(numberOne, numberTwo);
        mMainView.updateResult(result);
    }
}

 /*Used by Presenter to update the View with the result*/
interface MainView {
    void updateResult(int result);
}

 /* Business Logic / Application State */
class CalculatorModel {
    public int addNumbers(int numberOne, int numberTwo) {
        return numberOne + numberTwo;
    }
}
```

In this example, the *MainActivity* class represents the View, responsible for handling user interactions and displaying the results. The *MainPresenter* class represents the Presenter, responsible for handling user interactions, updating the Model, and updating the View with the result. The *CalculatorModel* class represents the Model, responsible for encapsulating the data and logic for the application. The *MainView*

interface provides the communication between the View and the Presenter. The **updateResult** method of the MainView interface is used by the Presenter to update the View with the result.

The presentation layer does not depend on the UI as it does in MVC and the presenter talks to the model layer through interfaces and gets model events back through interfaces as well. Therefore, the presenter will implement model layer interfaces and will communicate with model layer as the presenter gets model layer notifications you will create view data and pass it to the view. The view is rendered with the update model on the screen. Checkout the full code here [11] : MVP Example

A.1.3 Model - View - ViewModel (MVVM)

```
/* MainActivity would be quite as same as MVC but with the below changes */
    mMainViewModel = new ViewModelProvider(this).get(MainViewModel.class);
    //Inside the onclick listener
    mMainViewModel.addNumbers(numberOne, numberTwo);
    //Setting up an observer to sync the most updated state
    mMainViewModel.getResult().observe(this, new Observer<Integer>() {
        @Override
        public void onChanged(Integer result) {
            mResultTextView.setText(String.valueOf(result));
        }
    });


 /* Business Logic / Application State */
```

```
public class CalculatorModel {
    public int addNumbers(int numberOne, int numberTwo) {
        return numberOne + numberTwo;
    }
}

/* ViewModel:Communicates with Model, Updates Result, Displays in View.*/
    public class MainViewModel extends ViewModel {
        private MutableLiveData<Integer> mResult = new MutableLiveData<>();
        private CalculatorModel mCalculatorModel;

        public MainViewModel() {
            mCalculatorModel = new CalculatorModel();
        }

        public LiveData<Integer> getResult() {
            return mResult;
        }

        public void addNumbers(int numberOne, int numberTwo) {
            int result = mCalculatorModel.addNumbers(numberOne, numberTwo);
            mResult.setValue(result);
        }

    /* In the above code, the MainViewModel class represents the ViewModel
        while the CalculatorModel class represents the Model.
```

```
    The ViewModel is responsible for holding and managing UI-related data
        in a lifecycle-conscious way, which means that the data survives
        configuration changes such as screen rotations. The ViewModel acts
        as a bridge between the View (in this case, the MainActivity) and
        the Model (in this case, the CalculatorModel), providing data to
        the View and serving as a handler for UI actions initiated from the
        View.
    */
}
```

In this example, the *MainActivity* class represents the View, responsible for handling user interactions and displaying the results. The *MainViewModel* class represents the ViewModel, responsible for encapsulating the data and logic for the application, and providing the results to the View. The *CalculatorModel* class represents the Model, responsible for encapsulating the data and logic for the application. The **MutableLiveData<Integer>** object mResult represents the data that is displayed in the View. The Observer in the MainActivity class listens for changes in the mResult object and updates the View with the result. The addNumbers method in the MainViewModel class is called when the user interacts with the add button, and updates the mResult object with the result. Checkout the full code here [11] : MVVM Example

A.1.4 Model - View - Update (MVU)

```
/* MainActivity would be quite as same as MVC but with the below changes */
```

```
    mMainViewModel = new ViewModelProvider(this).get(MainViewModel.class);
    //Inside the onclick listener
    mModel.update(new Message.AddNumbers(numberOne, numberTwo));
    // Initialize the view by sending an initial message to the model.
    mModel.update(new Message.Initialize(this));

// The View receives messages from the model and updates the UI
     accordingly.
    private void updateView(Model.State state) {
        mResultTextView.setText(String.valueOf(state.result));
    }

    // The Model represents the current state of the app and updates the
        view by sending messages.
    private static class Model {
        private State mState = new State();
        // The update() method receives messages and updates the state
            accordingly.
        public void update(Message message) {
            mState = updateState(mState, message);
            updateView(mState);
        }
        // The updateState() method applies the message to the current
            state to produce a new state.
        private State updateState(State state, Message message) {
            if (message instanceof Message.Initialize) {
```

```
            state.result = 0;
            state.view = ((Message.Initialize) message).view;
        } else if (message instanceof Message.AddNumbers) {
            Message.AddNumbers addNumbers = (Message.AddNumbers) message;
            state.result = addNumbers.numberOne + addNumbers.numberTwo;
        }
        return state;
    }
    // The updateView() method sends a message to the View to update
        the UI.
    private void updateView(State state) {
        state.view.updateView(state);
    }
    // The State class represents the current state of the app.
    private static class State {
        public int result;
        public MainActivity view = null; //If I kept View instead on
            MainActivity is gave me a casting error on compile time
    }
}
// The Message class defines the different types of messages that can
    be sent from the View to the Model.
private static abstract class Message {

    public static class Initialize extends Message {
        public final MainActivity view;
```

```
            public Initialize(MainActivity view) {
                this.view = view;
            }
        }

        public static class AddNumbers extends Message {
            public final int numberOne;
            public final int numberTwo;

            public AddNumbers(int numberOne, int numberTwo) {
                this.numberOne = numberOne;
                this.numberTwo = numberTwo;
            }
        }
    }
}
```

In this example, the app has a single activity (**MainActivity**) that implements the View component of the MVU architecture. The Model component is represented by the **Model** class, which maintains the current state of the app and updates the view by sending messages. The **Update** function is split between the **Model.update()** and **Model.updateState()** methods. **Messages** are defined using the Message class, which defines the different types of messages that can be sent from the View to the Model. Each message can have different data associated with it. In this example, there are two types of messages: **Initialize**, which is sent when the app starts up, and **AddNumbers**, which is sent when the user clicks the "Add" button. *The same*

implementation is way easier and readable in Elm Programming. Checkout the full code here [11] : MVU Example

A.2 Android XML Layout vs Jetpack Compose

In the native Android code, we use an XML Layout file in which we design the components and the widgets to be shown to the user and they can interact with it. In this example, we will create a simple list view that consists of one text view and one button, when the user clicks on the button, it modifies the text to a new text.

```
/*Create a base adapter and inside the get view, you will see this inflate
    method*/
  if (convertView == null) {
        convertView =
            LayoutInflater.from(MainActivity.this).inflate(R.layout.list_item,
            parent, false);
        viewHolder = new ViewHolder();
        viewHolder.textView = convertView.findViewById(R.id.item_text);
        viewHolder.button = convertView.findViewById(R.id.item_button);
        convertView.setTag(viewHolder);
    } else {
        viewHolder = (ViewHolder) convertView.getTag();
    }

/*Set the adapter to the listview*/
    listView.setAdapter(adapter);
```

```
/*list_item.xml*/
    <?xml version="1.0" encoding="utf-8"?>
    <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:id="@+id/item_text"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"/>

        <Button
            android:id="@+id/item_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Modify"/>

    </LinearLayout>
```

In getView(), the method checks if convertView is null (indicating a new view needs to be inflated) or not null (indicating an existing view can be reused). If convertView is null, the layout listitem.xml is inflated using LayoutInflater.from(), and

the ViewHolder is created and attached to the converted view using setTag(). If convertView is not null, the existing ViewHolder is retrieved using getTag().The getView() method then sets the appropriate data (item text) and click listeners (for the "Modify" button) on the views within the ViewHolder.

We would manually create instances of each UI element in the imperative approach, and then make modifications to them directly thereby we must search for the views using their IDs and set the appropriate data on them repeatedly.

In Jetpack Compose, we use Compose's declarative UI approach to build the UI and manage the state. Assuming we have created the blueprint of the view that what we want to show to the user.

```
/*We use 'LazyColumn', a composable to display the list of items and
    'mutableStateListOf' to hold the list items*/
 val items = remember { mutableStateListOf<String>() }
    if (items.isNotEmpty()) {
    LazyColumn {
        items(items) { item ->
            ListItem(item) {
                val modifiedItem = item + `` Modified"
                items[items.indexOf(item)] = modifiedItem
            }
        }
    }
  } else ...
```

Instantly you would have noticed something, No XML layout file!

The App composable is the main entry point that manages the UI. It defines a mutableStateListOf to hold the items.

Inside the App composable, a Column composable is used to arrange the UI elements vertically. The "Add Item" button is displayed using the Button composable. When the button is clicked, a new item is added to the items list using the add() function. If the items list is not empty, a LazyColumn composable is used to display the list of items. The items list is iterated using the items() function, and for each item, the ListItem composable is invoked.

In the ListItem composable, when the "Modify" button is clicked, the onItemClick lambda is invoked. Inside the lambda, the item is modified by appending "Modified" to it, and the updated item is stored back in the items list at the appropriate index using the index obtained from items.indexOf(item).

By using Jetpack Compose, we can define the UI hierarchy and state management in a more concise and declarative manner. The code is simpler, more readable, and focuses on describing what the UI should look like and how it should behave rather than manually manipulating views and adapters.

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[2] M. Alpuente, R. Barbuti, and I. Ramos, editors. *1994 Joint Conference on Declarative Programming, GULP-PRODE 94 Peñiscola, Spain, September 19-22, 1994, Volume 2*, 1994.

[3] C. Anand and C. Schankula. GraphicSVG 7.2.0, 2017. URL `https://package.elm-lang.org/packages/MacCASOutreach/graphicsvg/latest/`. (Accessed on 06/26/2023).

[4] D. Andreou. Element Cost In Data Structures. URL `https://github.com/DimitrisAndreou/memory-measurer/blob/master/ElementCostInDataStructures.txt`. (Accessed on 07/08/2023).

[5] Apple. What's new in SwiftUI. URL `https://developer.apple.com/videos/play/wwdc2021/10018/`. (Accessed on 2023-06-14).

[6] Apple. Cocoa (Touch), April 2018. URL `https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Cocoa.html`. (Accessed on 06/12/2023).

[7] Apple. SwiftUI Overview, 2023. URL https://developer.apple.com/xcode/swiftui/. (Accessed on 06/12/2023).

[8] Apple. UIKit, 2023. URL https://developer.apple.com/documentation/uikit. (Accessed on 06/12/2023).

[9] Apple. Driving changes in your UI with state and bindings, 2023. URL https://developer.apple.com/tutorials/swiftui-concepts/driving-changes-in-your-ui-with-state-and-bindings. (Accessed on 06/11/2023).

[10] Apple. WWDC23, 2023. URL https://developer.apple.com/wwdc23/. (Accessed on 2023-06-14).

[11] A. Arumugasamy. Design Patterns Examples, 2023. URL https://github.com/FondationSTaBLFoundation/Widgets/tree/akshay_widgets/Design%20Patterns%20Examples. (Accessed on 07/15/2023).

[12] E. Baer. *What React Is and Why It Matters*. O'Reilly Media, Inc., August 2018. ISBN 9781491996737. URL https://www.oreilly.com/library/view/what-react-is/9781491996744/ch01.html. (Accessed on 06/15/2023).

[13] E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. D. Meuter. A Survey on Reactive Programming. *ACM Comput. Surv.*, 45(4), August 2013. doi: 10.1145/2501654.2501666.

[14] C. Barker. *Learn SwiftUI: An introductory guide to creating intuitive cross-platform user interfaces using Swift 5*. Packt Publishing Ltd, 2020.

[15] A.-C. Bellini and N. Butcher. Jetpack Compose is now 1.0, 2021. URL

https://android-developers.googleblog.com/2021/07/jetpack-compose-announcement.html. (Accessed on 06/12/2023).

[16] K. Bernales. 12 Absolute Principles of Material Design, April 2016. URL https://www.creative-tim.com/blog/web-design/12-absolute-principles-material-design/. (Accessed on 06/15/2023).

[17] A. P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet. *Pharo by example*. Lulu.com, 2010. ISBN 9783952334140.

[18] Bmbarbour. Defining 'Declarative', January 2012. URL https://awelonblue.wordpress.com/2012/01/12/defining-declarative/. (Accessed on 2023-06-06).

[19] Boformer. Answer to "Flutter: Mutable Fields in Stateless Widgets". URL https://stackoverflow.com/a/53192845. (Accessed on 2023-06-20).

[20] D. Browne, M. Norman, and E. Adhami. Methods for Building Adaptive Systems. In D. BROWNE, P. TOTTERDELL, and M. NORMAN, editors, *Adaptive User Interfaces*, pages 85–130. Academic Press. doi: https://doi.org/10.1016/B978-0-12-137755-7.50009-1.

[21] Y. Bugayenko and S. Zykov. The Impact of Object Immutability on the Java Class Size. *Procedia Computer Science*, 176:1868–1872, 2020.

[22] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. ISBN 0471958697.

[23] L. Byron. Immutable Data and React, 2015. URL `https://www.youtube.com/watch?v=I7IdS-PbEgI&ab_channel=MetaDevelopers`. (Accessed on 07/11/2023).

[24] L. Byron. Immutable.js, 2015. URL `https://immutable-js.com/#introduction`. (Accessed on 07/09/2023).

[25] M. Carlsson and T. Hallgren. *Fudgets: purely functional processes with applications to graphical user interfaces*. Number N.S., 1366 in Doktorsavhandlingar vid Chalmers Tekniska Högskola. Chalmers Univ. of Technology. ISBN 978-91-7197-611-6.

[26] M. Carlsson and T. Hallgren. Fudgets: A graphical user interface in a lazy functional language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, page 321–330, New York, NY, USA, 1993. Association for Computing Machinery. doi: 10.1145/165180.165228.

[27] J. Clarke and P. Gusmorino. Building amazing applications with the Fluent Design System, 2017. URL `https://learn.microsoft.com/en-us/events/connect-2017/b107`. (Accessed on 06/12/2023).

[28] M. Coblenz, J. Sunshine, J. Aldrich, B. Myers, S. Weber, and F. Shull. Exploring Language Support for Immutability. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 736–747, New York, NY, USA, 2016. Association for Computing Machinery. doi: 10.1145/2884781.2884798.

[29] R. C. Collins. Embracing Immutable Architecture, Sep 2016. URL https://medium.com/react-weekly/embracing-immutable-architecture-dc04e3f08543. (Accessed on 06/26/2023).

[30] W. Cook. Declarative versus Imperative. URL http://wcook.blogspot.com/2013/05/declarative-versus-imperative.html. (Accessed on 2023-06-06).

[31] G. H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Brown University, USA, 2008. AAI3335643.

[32] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell workshop*, pages 41–69, 2001.

[33] E. Czaplicki. Custom Types · An Introduction to Elm, . URL https://guide.elm-lang.org/types/custom_types.html. (Accessed on 06/22/2023).

[34] E. Czaplicki. The Elm Architecture · An Introduction to Elm, . URL https://guide.elm-lang.org/architecture/index.html. (Accessed on 06/21/2023).

[35] E. Czaplicki. Elm: Concurrent FRP for Functional GUIs. *Senior thesis, Harvard University*, 30, 2012.

[36] E. Czaplicki. A Farewell to FRP, 2016. URL https://elm-lang.org/news/farewell-to-frp. (Accessed on 06/21/2023).

[37] E. Czaplicki. Blazing Fast HTML, 2016. URL https://elm-lang.org/news/blazing-fast-html-round-two. [Accessed 11-Jun-2023].

[38] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. *ACM SIGPLAN Notices*, 48(6):411–422, 2013.

[39] N. D. A Complete Guide And Comparison Of MVC and MVVM, October 2021. URL `https://www.intuz.com/blog/guide-on-mvc-vs-mvvm`. (Accessed on 06/15/2023).

[40] 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. 270:18–36. doi: 10.4204/EPTCS.270.2.

[41] V. Davis. Apple releases native SwiftUI framework with declarative syntax, live editing, and support of Xcode 11 beta Packt Hub. URL `https://hub.packtpub.com/apple-releases-native-swiftui-framework-with-declarative-syntax-live-editing-and-support-of-xcode-11-beta/`. (Accessed on 06/12/2023).

[42] T. DeMarco. *Structured Analysis and System Specification*, page 409–424. Yourdon Press, USA, 1979. ISBN 0917072146.

[43] K. Destin. iOS Architecture Patterns. URL `https://sites.tufts.edu/eeseniordesignhandbook/files/2020/05/Destin_Maximum-Blue-Green-Tech-Note.pdf`.

[44] Doug Engelbart Institute. Firsts: The Demo, 2023. URL `https://dougengelbart.org/content/view/209/`. (Accessed on 06/12/2023).

[45] Dykraf. How to use Bootstrap UI Library in React.js Ecosystem, Dec

2022. URL https://dykraf.com/blog/bootstrap-with-reactstrap-and-react-bootstrap. (Accessed on 06/15/2023).

[46] Eclipsepedia. Is SWT better than Swing? URL https://wiki.eclipse.org/FAQ_Is_SWT_better_than_Swing%3F. (Accessed on 2023-06-15).

[47] C. Elliott and P. Hudak. Functional Reactive Animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional Programming*, ICFP '97, pages 263–273. Association for Computing Machinery. doi: 10.1145/258948.258973.

[48] Endoflife.date. Vue.js, June 2023. URL https://endoflife.date/vue. (Accessed on 06/15/2023).

[49] R. K. Eng. How learning Smalltalk can improve your skills as a programmer? URL https://techbeacon.com/app-dev-testing/how-learning-smalltalk-can-make-you-better-developer. (Accessed on 06/15/2023).

[50] R. K. Eng. Who uses Smalltalk?, December 2015. URL https://medium.com/smalltalk-talk/who-uses-smalltalk-c6fdaa6319a. (Accessed on 06/15/2023).

[51] R. K. Eng. The best way to teach children how to program is with a good teaching language, January 2017. URL https://richardeng.medium.com/i-believe-the-best-way-to-teach-children-how-to-program-is-with-a-good-teaching-language-e62ace56fa06. (Accessed on 06/15/2023).

[52] J. Evans. WWDC: Apple's call to code and the no-code future. URL

https://www.computerworld.com/article/3656908/wwdc-apples-call-to-code-and-the-no-code-future.html. (Accessed on 2023-06-14).

[53] M. Fayad and D. C. Schmidt. Object-Oriented Application Frameworks. *Commun. ACM*, 40(10):32–38, oct 1997. doi: 10.1145/262793.262798.

[54] R. Fleury. UI, State Mutation, Jank, and Hotkeys. URL https://www.rfleury.com/p/ui-part-8-state-mutation-jank-and. (Accessed on 06/23/2023).

[55] Flutter Agency. Stateful And Stateless Widget In Flutter, February 2022. URL https://flutteragency.com/relation-between-stateful-and-stateless-widgets-in-flutter/. (Accessed on 2023-06-20).

[56] FlutterMapp. Every Flutter Widget Explained!, Jan. 2023. URL https://www.youtube.com/watch?v=kj_tldMmu4w&ab_channel=FlutterMapp. (Accessed on 06/27/2023).

[57] J. D. Fokker, S. Holdermans, A. Löh, and S. Swierstra. Functional Programming, Sep 2011. URL https://docplayer.nl/10678104-Functional-programming.html. (Accessed on 06/15/2023).

[58] Fondation STaBL Foundation. ShapeCreator, 2023. URL https://macoutreach.rocks/SC3.html. (Accessed on 08/04/2023).

[59] M. Fowler. Design - Who needs an architect? *IEEE Software*, 20(5):11–13, Sep. 2003. ISSN 1937-4194. doi: 10.1109/MS.2003.1231144.

[60] M. Fowler. Inversion Of Control, June 2005. URL https://martinfowler.com/bliki/InversionOfControl.html. (Accessed on 06/20/2023).

[61] M. Fowler. GUI Architectures, July 2006. URL `https://martinfowler.com/eaaDev/uiArchs.html`. (Accessed on 2023-02-02).

[62] M. Fowler. Software Architecture Guide, August 2019. URL `https://www.martinfowler.com/architecture/`. (Accessed on 06/22/2023).

[63] M. Fowler's. Presentation Model, July 2004. URL `https://martinfowler.com/eaaDev/PresentationModel.html`. (Accessed on 06/22/2023).

[64] W. O. Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. Wiley Pub, 3rd ed edition. ISBN 978-0-470-05342-3. OCLC: ocm76792111.

[65] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. ISBN 978-0-201-63361-0.

[66] V. Gaudioso. MVVM: Model-View-ViewModel. In T. Brown, B. Renow-Clarke, C. Collins, C. Andres, S. Anglin, M. Beckner, E. Buckingham, G. Cornell, J. Gennick, J. Hassell, M. Lowman, M. Moodie, D. Parkes, J. Pepper, F. Pohlmann, D. Pundick, D. Shakeshaft, M. Wade, and T. Welsh, editors, *Foundation Expression Blend 4 with Silverlight*, pages 341–367. Apress. doi: 10.1007/978-1-4302-2974-2_15.

[67] C. Ghezzi and M. Jazayeri. *Programming language concepts, Third edition*. John Wiley & Sons, 1996.

[68] Google. Thinking in Compose-Jetpack Compose. URL `https://`

developer.android.com/jetpack/compose/mental-model. (Accessed on 06/12/2023).

[69] Google. Widgets library - Dart API. URL https://api.flutter.dev/flutter/widgets/widgets-library.html. (Accessed on 06/27/2023).

[70] Google. FAQ (Flutter), 2023. URL https://docs.flutter.dev/resources/faq#what-programming-paradigm-does-flutters-framework-use. (Accessed on 06/12/2023).

[71] Google. Flutter architectural overview, 2023. URL https://docs.flutter.dev/resources/architectural-overview. (Accessed on 06/12/2023).

[72] Google. Guide to app architecture Android Developers, 2023. URL https://developer.android.com/topic/architecture#single-source-of-truth. (Accessed on 06/12/2023).

[73] Google. Jetpack Compose UI App Development Toolkit - Android Developers, 2023. URL https://developer.android.com/jetpack/compose. (Accessed on 06/12/2023).

[74] Google. Start thinking declaratively, 2023. URL https://docs.flutter.dev/data-and-backend/state-mgmt/declarative. (Accessed on 06/11/2023).

[75] Google, Inc. StatelessWidget class. URL https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html. (Accessed on 2023-05-09).

[76] Google Trends. Flutter, React Native, 2023. URL

https://trends.google.com/trends/explore?date=all&q=%2Fg%2F11f03_rzbg,%2Fg%2F11h03gfxy9. (Accessed on 06/15/2023).

[77] Google/Guava. ImmutableCollectionsExplained·. URL https://github.com/google/guava/wiki/ImmutableCollectionsExplained. (Accessed on 07/08/2023).

[78] M. D. Griffith. Elm-ui 1.1.8. URL https://package.elm-lang.org/packages/mdgriffith/elm-ui/latest/. (Accessed on 2023-06-15).

[79] C. Guindon. SWT: The Standard Widget Toolkit. URL https://www.eclipse.org/swt/. (Accessed on 2023-06-15).

[80] P. Haller and L. Axelsson. Quantifying and Explaining Immutability in Scala. *Electronic Proceedings in Theoretical Computer Science*, 246:21–27, April 2017. doi: 10.4204/eptcs.246.5.

[81] T. Hallgren and M. Carlsson. Fudgets. URL http://www.altocumulus.org/Fudgets/. (Accessed on 06/14/2023).

[82] HaskellWiki Community Contributors. Referential transparency. URL https://wiki.haskell.org/Referential_transparency#notes. (Accessed on 06/23/2023).

[83] HaskellWiki Community Contributors. Parametric Polymorphism, January 2015. URL https://wiki.haskell.org/Polymorphism#Parametric_polymorphism. (Accessed on 06/26/2023).

[84] M. Hevery. AngularJS — Superheroic JavaScript MVW Framework, 2021. URL https://angularjs.org/. (Accessed on 06/15/2023).

[85] M. Hevery. Angular - Introduction to the Angular docs, 2023. URL `https://angular.io/docs`. (Accessed on 06/15/2023).

[86] R. Hickey. Clojure - Concurrent Programming, 2008-2022. URL `https://clojure.org/about/concurrent_programming`. (Accessed on 06/23/2023).

[87] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Reactive Programming. In J. Jeuring and S. L. P. Jones, editors, *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*, pages 159–187. Springer Berlin Heidelberg. doi: 10.1007/978-3-540-44833-4_6.

[88] P. Hudson. SwiftUI vs UIKit – comparison of building the same app in each framework. URL `https://www.youtube.com/watch?v=qk2y-TiLDZo`. (Accessed on 2023-06-12).

[89] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32 (2):98—107, 1989. doi: 10.1093/comjnl/32.2.98.

[90] L. Jeroen. UIKit vs. SwiftUI - Choosing the Right Framework!, Jan 2022. URL `https://getstream.io/blog/uikit-vs-swiftui/`. (Accessed on 06/12/2023).

[91] Jessta. Messages purpose - Learn - Elm, 2021. URL `https://discourse.elm-lang.org/t/messages-purpose/6778/3`. (Accessed on 06/15/2023).

[92] A. C. Kay. The Early History Of Smalltalk. URL `http://worrydream.com/EarlyHistoryOfSmalltalk/`. (Accessed on 2023-06-08).

[93] B. Kaya. What are the reasons why object oriented programming is not used by JavaScript? What are the alternatives that can be used instead of it (for example functional, procedural)?, February 2022. URL https://www.quora.com/What-are-the-reasons-why-object-oriented-programming-is-not-used-by-JavaScript-What-are-the-alternatives-that-can-be-used-instead-of-it-for-example-functional-procedural. (Accessed on 06/15/2023).

[94] T. Kestermann. The Standard Widget Toolkit (SWT) — Java UI at its Best. URL https://medium.com/@TorstenKestermann/the-standard-widget-toolkit-swt-java-ui-at-its-best-part-1-444f7f15b74. (Accessed on 2023-06-15).

[95] Kindsonthegenius. Elm – Custom Types (Algebraic Data Types), 2022. URL https://kindsonthegenius.com/elm/elm-custom-types-algebraic-data-types/. (Accessed on 06/22/2023).

[96] Kofi Group. 7 reasons why VueJS is so popular. URL https://www.kofi-group.com/7-reasons-why-vuejs-is-so-popular/. (Accessed on 2023-06-15).

[97] A. S. Korban. Elm-ui: The CSS Escape Plan. URL https://korban.net/elm/elm-ui-guide/. (Accessed on 2023-06-15).

[98] W. Kreutzer. Basic Aspects of Squeak and the Smalltalk-80 Programing Language, 1998.

[99] S. Kumar. Data Flow Diagrams and Data Dictionaries. URL `https://www.scaler.com/topics/data-flow-diagrams/`. (Accessed on 06/28/2023).

[100] J. Kunasaikaran and A. Iqbal. A brief overview of functional programming languages. *Electronic Journal of Computer Science and Information Technology*, 6(1), 2016.

[101] E. Kuzmenko. Model-View-Controller Architecture Pattern: Usage, Advantages, Examples, June 2022. URL `https://hackernoon.com/model-view-controller-architecture-pattern-usage-advantages-examples`. (Accessed on 06/15/2023).

[102] D. Larkin, M. Morse, J. Neider, and C. Rose. NeXTstep Concepts, pp. 4.36—4.45, NeXT Computer. *Inc., Redwood City, CA*, 1990.

[103] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sep. 1987. ISSN 1558-2256. doi: 10.1109/PROC.1987.13876.

[104] D. Leijen. WxHaskell: a portable and concise GUI library for haskell. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 57–68. Association for Computing Machinery. doi: 10.1145/1017472.1017483.

[105] M. Manjunath. AngularJS and Angular 2+: A Detailed Comparison, April 2018. URL `https://www.sitepoint.com/angularjs-vs-angular/`. (Accessed on 06/15/2023).

[106] W. L. Martinez. Graphical User Interfaces. *WIREs Computational Statistics*, 3(2):119–133, 2011. doi: https://doi.org/10.1002/wics.150.

[107] Material-UI Contributors. Material-UI: A popular React UI framework, 2023. URL https://v4.mui.com/. (Accessed on 06/15/2023).

[108] R. M. Max Goldman. Mutability & Immutability, Fall 2015. URL https://web.mit.edu/6.005/www/fa15/classes/09-immutability/. (Accessed on 06/26/2023).

[109] J. McCormack and P. Asente. An Overview of the X Toolkit. In *Proceedings of the 1st Annual ACM SIGGRAPH Symposium on User Interface Software*, UIST '88, page 46–55, New York, NY, USA, 1988. Association for Computing Machinery. doi: 10.1145/62402.62407.

[110] J. McLaughlin. Fluent: Design Behind the Design. How our Fluent Design System focuses, May 2019. URL https://medium.com/microsoft-design/fluent-design-behind-the-design-973028062fcc. (Accessed on 06/12/2023).

[111] Microsoft. Office UI Fabric JS. URL https://developer.microsoft.com/en-us/fabric-js. (Accessed on 06/12/2023).

[112] Microsoft. Get started - Fluent UI, 2023. URL https://developer.microsoft.com/en-us/fluentui#/. (Accessed on 06/12/2023).

[113] Microsoft Fluent Wiki. Integrating Fluent — Overview, June 2020. URL https://github.com/projectfluent/fluent/wiki/Integrating-Fluent-%E2%80%94-Overview. (Accessed on 06/11/2023).

[114] B. Moseley and P. Marks. Out of the tar pit. *Software Practice Advancement (SPA)*, 2006.

[115] Mozilla.org contributors. Element - MDN Web Docs, 2023. URL https://developer.mozilla.org/en-US/docs/Glossary/Element. (Accessed on 06/22/2023).

[116] Mozilla.org contributors. HTML - MDN Web Docs, 2023. URL https://developer.mozilla.org/en-US/docs/Glossary/HTML. (Accessed on 06/22/2023).

[117] Mozilla.org contributors. Introduction to the DOM, 2023. URL https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction. (Accessed on 06/22/2023).

[118] M. Muccinelli. Flutter: Dart Immutable Objects and Values. URL https://levelup.gitconnected.com/flutter-dart-immutable-objects-and-values-5e321c4c654e. (Accessed on 2023-05-12).

[119] A. Mukherjee. Component Communication in Angular (Parent to Child & Child to Parent) - DEV Community, Jan 2022. URL https://dev.to/this-is-angular/component-communication-parent-to-child-child-to-parent-5800. (Accessed on 07/05/2023).

[120] R. Muliyashiya. Android MVVM Patterns with LiveData, April 2023. URL https://devblog.link/android-architecture-patterns-mvvm-and-live-data/. (Accessed on 06/15/2023).

[121] B. Myers, S. E. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, 2000.

[122] B. A. Myers. User interface software tools. 2(1):64–103. doi: 10.1145/200968.200971. (Accessed on 2023-06-07).

[123] B. A. Myers. *A foundation for user interface construction*. CRC Press, 1992.

[124] R. Noble and C. Runciman. *Lazy functional components for graphical user interfaces*. PhD thesis, Citeseer, 1995.

[125] D. A. Norman. *The Design of Everyday Things*. Basic Books, Inc., USA, 2002. ISBN 9780465067107.

[126] S. Northover and C. MacLeod. Writing your own widget, March 2001. URL `https://www.eclipse.org/articles/Article-Writing%20Your%20Own%20Widget/Writing%20Your%20Own%20Widget.htm`. (Accessed on 2023-06-15).

[127] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: Updated for Scala 2.12*. Artima Incorporation, Sunnyvale, CA, USA, 3rd edition, 2016. ISBN 0981531687.

[128] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[129] Oracle. Lesson: Getting Started with Swing. URL `https://docs.oracle.com/javase/tutorial/uiswing/start/index.html`. (Accessed on 2023-06-15).

[130] Oracle. Abstract Window Toolkit (AWT), 2023. URL `https://docs.oracle.com/javase/8/docs/technotes/guides/awt/`. (Accessed on 06/15/2023).

[131] Oracle. Open JavaFX, 2023. URL https://openjfx.io/index.html. (Accessed on 2023-06-15).

[132] oTree Contributors. Forms, 2023. URL https://otree.readthedocs.io/en/latest/forms.html. (Accessed on 06/15/2023).

[133] M. Pawlan. What Is JavaFX?, April 2013. URL https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm. (Accessed on 06/15/2023).

[134] A. J. Perlis. Special Feature: Epigrams on Programming. *SIGPLAN Not.*, 17 (9):7–13, sep 1982. doi: 10.1145/947955.1083808.

[135] Pharo community. *Pharo-Flyer-cheat-sheet*. URL https://files.pharo.org/media/flyer-cheat-sheet.pdf. (Accessed on 06/15/2023).

[136] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[137] H. H. Porter III. Smalltalk: A white paper overview, 2003.

[138] M. Potel. MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java. 1996. URL https://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

[139] P. Poudel. Beginning Elm, 2018. URL https://elmprogramming.com/pattern-matching.html. (Accessed on 06/29/2023).

[140] Pub.dev. Cupertino_stepper (Flutter Package). URL https://pub.dev/packages/cupertino_stepper. (Accessed on 06/27/2023).

[141] W. V. O. Quine. *Word and Object*. The MIT Press. doi: 10.7551/mitpress/9636.001.0001.

[142] J. Rambhia. Introduction to MVVM architecture in Android, July 2019. URL https://jayrambhia.com/blog/android-mvvm-intro. (Accessed on 07/07/2023).

[143] ReactEurope. Immutable application architecture - lee byron, 2018. URL https://www.youtube.com/watch?v=oTcDmnAXZ4E&list=PLCC436JpVnK1X7atG6EIz467Evs4TMX_5&index=12&ab_channel=ReactEurope. (Accessed on 06/26/2023).

[144] U. Reddy and J. Chen. Functional programming - What is referential transparency?, 2019. URL https://stackoverflow.com/questions/210835/what-is-referential-transparency/9859966. (Accessed on 07/31/2023).

[145] M. Resnick. ScratchJr. URL http://www.scratchjr.org/. (Accessed on 06/15/2023).

[146] H. Rich. Simple Made Easy. URL https://github.com/matthiasn/talk-transcripts/blob/master/Hickey_Rich/SimpleMadeEasy.md. (Accessed on 07/07/2023).

[147] M. Sage. FranTk - a declarative GUI language for Haskell. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 106–117. Association for Computing Machinery. doi: 10.1145/351240.351250.

[148] D. Santos. Evolve Unidirectional Data Flow a.k.a MVI into MVVM + Jetpack Compose, Dec 5 2021. URL https://medium.com/eureka-engineering/

evolve-unidirectional-data-flow-a-k-a-mvi-into-mvvm-jetpack-compose-e6858a767290. (Accessed on 07/07/2023).

[149] Scratch Foundation. Scratch - Imagine, Program, Share. URL https://scratch.mit.edu/. (Accessed on 06/15/2023).

[150] N. Sculthorpe. *Towards safe and efficient functional reactive programming*. PhD thesis, University of Nottingham, 2011.

[151] Self Language Team. Morphic, 2018. URL https://wiki.squeak.org/squeak/30. (Accessed on 2023-06-16).

[152] Siemens. Low-Code Achieves Mainstream Status According to New Ground-Breaking Research. URL https://www.mendix.com/press/low-code-achieves-mainstream-statusaccording-to-new-ground-breaking-research/. (Accessed on 2023-06-14).

[153] M. Skoczylas. The idea behind functional programming. URL https://pragmaticreview.com/the-idea-behind-functional-programming/. (Accessed on 2023-06-06).

[154] Software Engineering Standards Committee of the IEEE Computer Society. *IEEE Recommended practice for architectural description of software-intensive systems - IEEE Std 1471-2000*. IEEE, 2000. URL http://cabibbo.dia.uniroma3.it/ids/altrui/ieee1471.pdf. (Accessed on 06/15/2023).

[155] M. O. Source. Introduction to React Native, 2023. URL https://reactnative.dev/docs/getting-started. (Accessed on 06/15/2023).

[156] M. O. Source. Quick Start – React, 2023. URL `https://react.dev/learn`. (Accessed on 06/15/2023).

[157] Stack Overflow. Stack Overflow Trends, 2023. URL `https://insights.stackoverflow.com/trends?tags=r%2Cstatistics`. (Accessed on 06/14/2023).

[158] K. A. Stokke, M. Barash, and J. Järvi. The Ultimate GUI Framework: Are We There Yet? In R. Lämmel, P. D. Mosses, and F. Steimann, editors, *Eelco Visser Commemorative Symposium (EVCS 2023)*, volume 109 of *Open Access Series in Informatics (OASIcs)*, pages 25:1–25:9, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/OASIcs.EVCS.2023.25.

[159] R. R. Swick and M. S. Ackerman. The X Toolkit: More Bricks for Building User-Interfaces or Widgets for Hire. In *Usenix Winter*, volume 88, pages 221–228. Citeseer, 1988.

[160] R. Trinkle. Overview — Reflex 0.5 documentation, 2018. URL `https://docs.reflex-frp.org/en/latest/overview.html`. (Accessed on 06/14/2023).

[161] R. Trinkle. Reflex: Higher-order Functional Reactive Programming, July 2023. URL `https://hackage.haskell.org/package/reflex`. (Accessed on 08/01/2023).

[162] K. Ubah. React.js vs React Native – What's the Difference?, Feb 2023. URL `https://www.freecodecamp.org/news/react-js-vs-react-native-whats-the-difference/`. (Accessed on 06/15/2023).

[163] B. Vyas. What's New in Angular 15? New Features and Updates, 2023. URL https://www.clariontech.com/blog/angular-15-with-new-features-and-updates. (Accessed on 06/15/2023).

[164] D. A. Watt. *Programming Language Design Concepts*. John Wiley Sons, Inc., Hoboken, NJ, USA, 2004.

[165] M. Weststrate. Introducing Immer: Immutability the easy way, January 2018. URL https://hackernoon.com/introducing-immer-immutability-the-easy-way-9d73d8f71cb3. (Accessed on 07/09/2023).

[166] Wikipedia contributors. Model–view–viewmodel, . URL https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93viewmodel&oldid=1157628301. (Accessed 22-June-2023).

[167] Wikipedia contributors. Standard Widget Toolkit, . URL https://en.wikipedia.org/w/index.php?title=Standard_Widget_Toolkit&oldid=1158028697. (Accessed on 2023-06-15).

[168] Wikipedia contributors. Flutter (software), 2023. URL https://en.wikipedia.org/wiki/Flutter_(software). (Accessed on 06/12/2023).

[169] Wikipedia contributors. Model–view–controller, 2023. URL https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller. (Accessed on 06/15/2023).

[170] Wikipedia contributors. React Native, 2023. URL https://en.wikipedia.org/wiki/React_Native. (Accessed on 06/15/2023).

[171] Wikipedia contributors. Side effect (computer science), 2023. URL https://en.wikipedia.org/w/index.php?title=Side_effect_(computer_science)&oldid=1142347657. (Accessed 23-June-2023).

[172] Wikipedia contributors. Software design pattern, 2023. URL https://en.wikipedia.org/w/index.php?title=Software_design_pattern&oldid=1161260019. (Accessed 21-June-2023).

[173] Wikipedia contributors. Abstract Window Toolkit, 2023. URL https://en.wikipedia.org/w/index.php?title=Abstract_Window_Toolkit&oldid=1158027564. (Accessed on 2023-06-15).

[174] Wikipedia contributors. JavaFX, 2023. URL https://en.wikipedia.org/w/index.php?title=JavaFX&oldid=1150518918. (Accessed on 2023-06-15).

[175] Wikipedia contributors. Swing (Java), 2023. URL https://en.wikipedia.org/w/index.php?title=Swing_(Java)&oldid=1148203815. (Accessed on 2023-06-15).

[176] F. V. Woka. The Difference between Stateless and Stateful Widgets in Flutter. URL https://blog.logrocket.com/difference-between-stateless-stateful-widgets-flutter/. (Accessed on 2023-06-20).

[177] S. M. Yacoub and H. H. Ammar. Toward Pattern-Oriented Frameworks. URL https://adtmag.com/articles/2001/07/16/toward-patternoriented-frameworks.aspx. (Accessed on 07/25/2023).

[178] E. You. Introduction to Vue.js, 2023. URL `https://vuejs.org/guide/introduction.html#what-is-vue`. (Accessed on 06/15/2023).

[179] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 75–84, Dubrovnik, Croatia, Sept. 2007.

[180] H. Zub. Why concept of immutability is so awfully important for a beginner front-end developer? URL `https://itnext.io/why-concept-of-immutability-is-so-damn-important-for-a-beginner-front-end-developer-8da85b565c8e`. (Accessed on 2023-04-17).