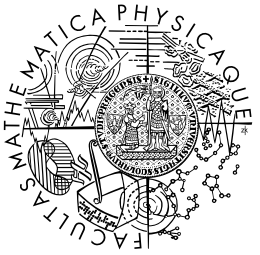

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jaroslav Švarc

Data-driven low-code programming system

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Tomáš Petříček, Ph.D.

Study programme: Computer science

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I want to thank my thesis advisor, Mgr. Tomáš Petříček Ph.D., for his invaluable guidance and support during the writing of this thesis. I also want to thank my family for supporting me throughout my studies.

Title: Data-driven low-code programming system

Author: Jaroslav Švarc

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Tomáš Petříček, Ph.D., Department of Distributed and Dependable Systems

Abstract: Software development using low-code programming systems is increasingly popular. Traditional low-code development systems that provide UI creation functionality primarily provide the UI-to-data approach, where developers create user interface elements before populating them with data. However, the data-to-UI approach, where the development process begins with concrete data that drives the creation of corresponding UI elements, remains unexplored as a primary development method. We present the InterfaceSmith prototype programming system, which implements data-to-UI as the primary development method for creating web applications' UI elements. We demonstrate how this data-driven approach enables incremental UI creation through a hole-based approach, where the system creates a UI skeleton using holes as placeholders that developers can incrementally replace with automatically generated UI elements based on the underlying data. The system aims to aid developers in modifying the interface through context menus and generates applications following the Elm architecture. Our evaluation through benchmarks, including a TO-DO list application and tasks from the 7GUIs benchmark suite, demonstrates the system's effectiveness in reducing the amount of code developers need to write while maintaining the ability to implement custom web application functionality.

Keywords: low-code programming, programming systems, data-driven programming

Název práce: Data-driven low-code programming system

Autor: Jaroslav Švarc

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Vývoj softwaru pomocí systémů, které minimalizují potřebu psaní velkého množství kódu (low-code), roste na popularitě. Tradiční low-code vývojové systémy obvykle umožňují vytvářet uživatelské rozhraní prostřednictvím přístupu „UI-to-data“, kdy vývojáři nejprve navrhnou prvky uživatelského rozhraní, které jsou následně naplněny daty. Oproti tomu přístup „data-to-UI“, při kterém vývoj začíná konkrétními daty a ta určují podobu odpovídajících prvků uživatelského rozhraní, zůstává jako hlavní metoda vývoje dosud málo prozkoumána. V této práci představujeme prototyp programovacího systému jménem InterfaceSmith, který implementuje přístup „data-to-UI“ jako hlavní metodu pro návrh uživatelských rozhraní webových aplikací. Tento přístup umožňuje postupné vytváření uživatelského rozhraní prostřednictvím konceptu „děr“ (holes), které slouží jako zástupné prvky. Ty mohou vývojáři postupně nahrazovat automaticky generovanými komponentami uživatelského rozhraní na základě dostupných dat. Systém si dává za cíl usnadnit úpravy rozhraní pomocí kontextových menu a umožnit generování aplikací podle architektury Elm. Testování systému na několika benchmarcích, jako je aplikace typu TO-DO list a úlohy z benchmarkové sady 7GUIs, demonstruje jeho schopnost snížit množství kódu, které vývojáři musí psát, aniž by tím byla omezena možnost implementace vlastní funkcionality webových aplikací.

Klíčová slova: low-code programování, vývojové nástroje, datově řízené programování

Contents

Introduction

Since the advent of the *World Wide Web*, developers have been creating web applications mainly by writing and modifying text-based code. Today, this process is made more accessible by using different tools, libraries, frameworks, and programming systems. For example, tools such as the *React* [1] library or the *Vue.js* [2] framework make it possible to compose web applications from reusable components but still require the user to write and modify text-based code.

Some programming systems allow an alternative approach of *low-code* software development. Pinho et al. [3] define *low-code programming systems* as "systems that aim to significantly reduce the amount of code needed to create specific applications." These systems employ an alternative way of creating and modifying the software components by providing a *Graphical User interface (GUI)* and modification tools that aid the user during the creation process. The popularity of these systems has been increasing in recent years, as stated by Sahay et al. [4].

Systems such as *Mendix* [5] or *Darklang* [6] specialize in providing a comprehensive low-code development platform for creating specific components of web applications. Other programming systems, such as *Sketch-n-Sketch* presented by Hempel et al. [7], combine the low-code approach with other programming techniques, such as *output-directed programming* [8], to enhance the system's functionality and development process.

Sahay et al. [4] identify that low-code systems provide two main approaches to creating the *User Interface (UI)* of software applications, and they are **UI to data** and **data to UI**. *UI to data* involves creating the application's UI and populating it with data, whereas *data to UI* is a data-driven approach where the UI is created *after* modeling necessary data. The *AppForge* programming system presented by Yang et al. [9] provides tools to create user interface elements of web applications and automatically creates corresponding database schema and application logic. The system also provides tools to create UI elements based on existing database schemas and custom logic, however the *UI to data* approach is still the main development method.

But what if we used the data-to-UI approach as the primary means of creating UI elements of client-side web applications? In this thesis, we aim to explore the *data to UI* approach in the context of creating *client-side web applications*, providing an alternative to traditional text-based development methods. We will design and implement a *prototype* low-code programming system named **InterfaceSmith** inspired mainly by the functionality of the *AppForge* programming system. The *data to UI* process will be influenced by the *type* of data provided to the system and also its *structure*. The application's UI elements will then be created through a sequence of steps, mirroring the structure of the provided data. The programming system aims to aid the developer in modifying the user interface by providing relevant context menus. After creating and modifying the UI elements, the system will provide behavior modification options inspired by *Elm* [10] and the *Elm architecture* [11]. After the developer finishes the implementation process, they can preview the created application and download its text-based representation.

Based on the definition of low-code programming systems, we will evaluate the results based on the number of lines of code (LOC) that the user needs to write to

implement the selected tasks and whether the functionality of the desired task has been successfully implemented.

InterfaceSmith

The main output of this thesis is the InterfaceSmith application shown in Figure 1. The user interface consists of a movable canvas containing various draggable canvas elements. The main element is the UI modification and preview element, which previews the application's created parts alongside modification menus. It is also resizable, simulating a browser window. Other elements include a view of the uploaded data, elements for creating and modifying the behavior of the UI elements, and elements for creating custom JavaScript functions. The resulting code is generated based on the created elements. The system creates an application in pure JavaScript using the Elm architecture[11], but it could be extended to target other technologies. We will describe the system's design, implementation, and functionality in the following chapters.

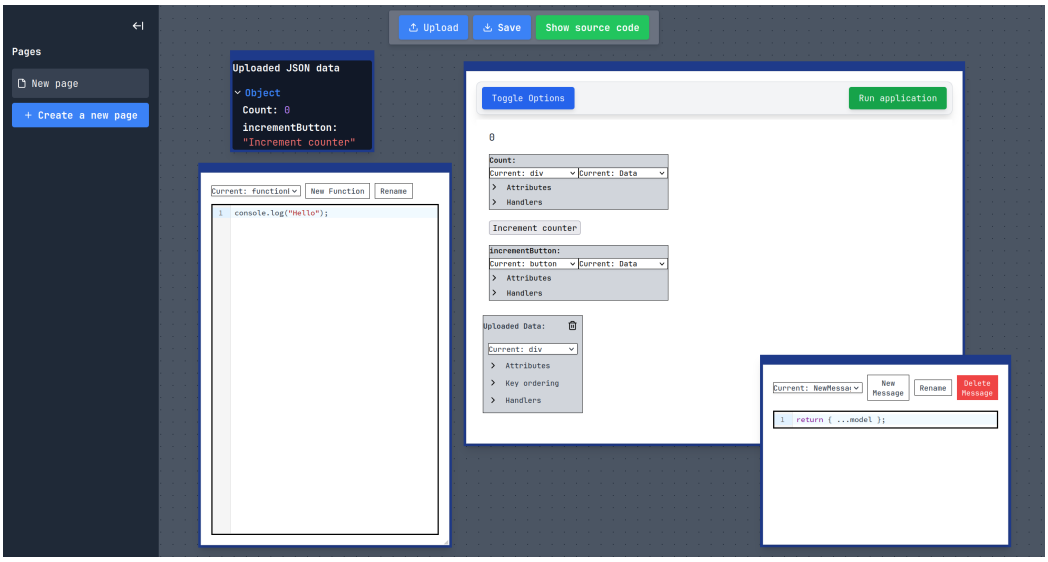

Figure 1 InterfaceSmith's UI example

Goals

The main goals of this thesis are the following:

1. Explore the applicability of the low-code programming system employing the data-driven approach to the creation of UI elements of web applications:
    - Define the *data-driven* approach.
    - Define the approach in the context of creating web application's UI elements based on concrete data.
    - Describe the design principles behind the approach.

- Present a description of creating UI elements based on the input data.
- Evaluate the feasibility of this approach in the context of the prototype application.

2. Create a working **prototype programming system** implementing the data-driven approach according to the design principles described in Chapter 2.
3. Benchmark the prototype application on the following tasks:
    - A simple **TO-DO list** application inspired by the *TodoMVC* [12].
    - **Counter** task from the *7GUIs benchmark* [13].
    - **Temperature converter** task from the *7GUIs benchmark* [13].

Contributions and Outline

This thesis contributes to the field of software development, specifically web development systems employing a data-driven low-code approach.

- In Chapter 2, we describe the main design principles that influenced the design of the prototype programming system.
- In Chapter 3, we describe the core logic of our programming system. We introduce the data-driven hole-based approach to creating UI elements, describe the domain model and essential operations, and describe the incremental creation process.
- In Chapter 4, we describe the implementation specifics of the *InterfaceSmith* application, including its architecture, technologies used, and the design of its user interface.
- In Chapter 5, we showcase the creation of a user interface for a simple TO-DO list application and several other tasks from *7GUIs benchmark* [13].
- In Chapter 6, we discuss the benefits and limitations of our proposed programming system, and we evaluate our *InterfaceSmith* application. We also provide several future avenues of research.

1 Background

1.1 Low-code development systems

In the previous chapter, we briefly described that our thesis aims to create a *low-code data-driven* programming system. In this section, we describe several existing systems that employ the low-code development approach to understand better how our resulting application fits into the low-code programming system landscape. The low-code development approach can be defined as follows:

Definition 1 (Low-code development approach)**.** *Defined by Pinho et al. [3] as "An approach for software development that uses tools that minimize (or eliminate) the number of lines of code written."*

This definition accommodates a broad spectrum of tools and development systems. The main category of these tools is *visual programming tools* such as *AppForge* [9], *Darklang* [6] or *Mendix* [5]. The visual programming tools provide a Graphical User Interface (GUI), which allows users to create and modify software by interacting with the interface. They differ in the style of UI, provided functionality, their intended use case, and other features, such as automatic deployment of the created software.

The next category of low-code development tools is *Integrated development environments (IDEs)* with code generation capabilities. For example, *JetBrains Rider* [14] provides advanced context-aware code auto-completion alongside tools to generate boilerplate code and project scaffolding. Code generation can also be provided by integrating *Large Language Models (LLMs)* into existing IDEs, such as the *Github Copilot* [15] extension for the *Visual Studio Code* [16] editor.

The last large category of low-code development tools is *Static site generators (SSGs)*, which transform files written in a simple markup language into a static website. A popular example of SSGs is the *Hugo* [17] static site generator, which also provides tools for customizing the transformation process and supports custom templating.

To better understand the *InterfaceSmith* system, we will describe four programming systems that greatly influence its design. The *Hypercard*, *AppForge*, *Darklang*, and *Sketch-and-Sketch* belong to the category of visual programming tools. Each system has a different intended use case, User Interface, and capabilities. In Chapter 2, we explain *how* these particular systems inspire the design of our system.

1.1.1 Hypercard

Hypercard is a low-code development system created by Bill Atkinson for the Macintosh operating system. Apple released the program in 1987 at the Macworld exposition in Boston [18]. Apple developed and maintained the program until 1998. The popularity of the program ensured that similar programs and clones of Hypercard were created after its discontinuation.

The following summary of functionality is based on a manual for the Hypercard system written by Goodman [19]. The fundamental elements that the user creates are called *cards*. Cards can hold data as text, have custom graphics, contain buttons, and implement custom behavior. Users can implement the custom behavior using a scripting language called *HyperTalk*. Then, users can group cards into *stacks*. A stack is

a collection of cards with the same type of information. The program saves a stack as a single file to the disk. Finally, users can distribute and modify these stacks. Creation and modification of the card is done through the low-code interface. The program provides several options for the different card elements.

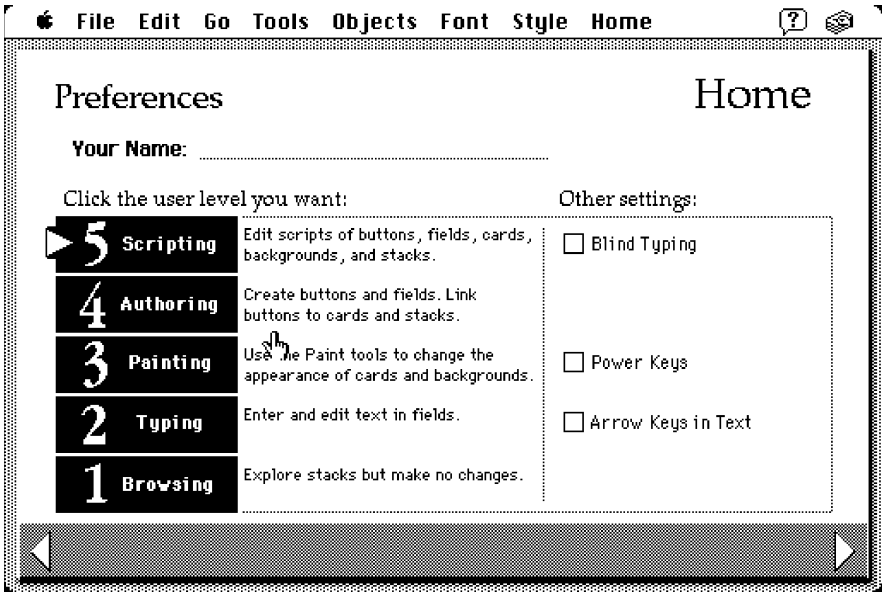

Figure 1.1 Selection of a user level in the Hypercard preferences menu. (Image created using a Hypercard emulator [20].)

The program offers users a choice of a user level, as seen in figure 1.1. The program changes the user interface's capabilities based on the selected user level. The higher the selected user level, the more options the program enables and displays. A selected level enables all previous-level options alongside other options and functionality. Users can choose from five different user levels:

1. **Browsing:** Enables no modification of cards and is mainly intended to be used to view the different cards inside a stack.
2. **Typing:** Enables inserting and modifying text data inside the cards.
3. **Painting:** Enables the creation of custom graphics inside the cards. The program provides different graphical options and tools.
4. **Authoring:** Adds the ability to add buttons and fields. This way, the users can add card functionality without writing code.
5. **Scripting:** Adds the ability to use the HyperTalk scripting language to modify the behavior of the different elements inside the cards.

This allows the system to be adopted by a wide range of users, as less experienced users can learn to use the interface more easily and gradually increase their proficiency with the program without being overwhelmed by options. Advanced users can set the highest user level and use the program in its entirety from the start.

1.1.2 AppForge

AppForge is a low-code programming system presented by Yang et al. [9]. It is designed as a "what you see is what you get" (abbreviated as *WYSIWYG*) platform for creating web applications using a low-code interface. The system provides options for creating UI elements and automatically generates the necessary parts of a functional web application, such as the underlying database schemas and application logic [9].

The users create UI elements using a low-code interface consisting of different menus, which we see in Figure 1.2. The users manually create the two main types of elements, and they are *Forms* and *Views*. Forms are used to add new data to the schema of a corresponding entity, and the Views are elements that display data from one or multiple database entities, and the user can modify how the data is displayed. The system automatically creates a new database table for each new entity or when the user creates a View which displays data from multiple entities.

Figure 1.2 An example of AppForge's UI by Yang et al. [9]

1.1.3 Darklang

Darklang [6] is a cloud-based low-code programming system for building web application backends created by Paul Biggar. The original system's development was discontinued in 2023 and renamed Darklang-classic.

The following summary follows the official Darklang documentation[1]. The Darklang project consists of a low-code editor and a programming language. It also provides tools for creating persistent storage and for deployment. The editor provides the service's main functionality. It consists of a canvas that displays the created application elements as draggable boxes as seen in Figure 1.3. Each box provides options to modify

[1] `https://docs.darklang.com/` [visited on 2024-04-22][online]

the element. These options include different menus and input fields. Some elements can be modified using the Darklang programming language to implement custom behavior. The user can create program elements of the following categories:

- HTTP handlers - definition of API endpoints
- Persistent storage - database creation and modification
- Workers - processing of background tasks
- Cron jobs - scheduled jobs with custom behavior
- REPL - a general-purpose programmable element
- Functions - a reusable element with custom behavior

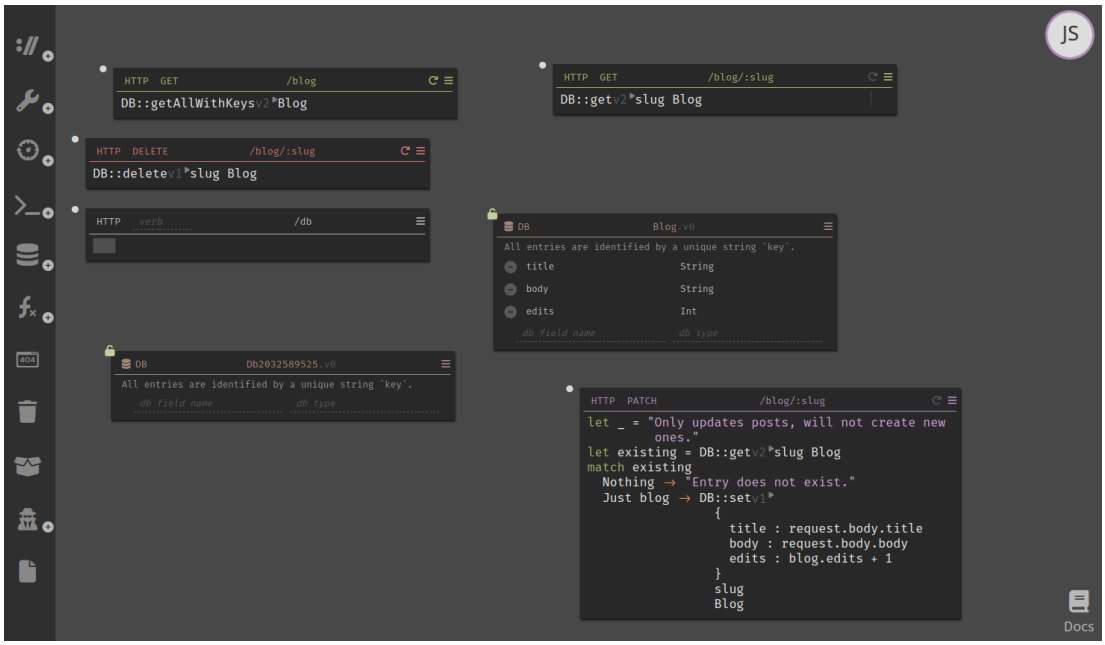

Figure 1.3 Example of Darklang's drag-and-drop user interface. (Image created using the Darklang-classic application [6].)

By combining these elements, the users can create application backends of varying complexity. The system automatically redeploys the entire application after the user creates and modifies a particular element. This allows for an interactive style of development where changes made to the system are almost immediately visible.

1.1.4 Sketch-and-Sketch - Output-directed programming

Output-directed programming is described by Hempel et al. [7] as a programming paradigm where users construct plain text programs using mouse-based operations on the program's output. The difference between this paradigm and other low-code programming approaches is that the source code is still the main representation of the program. When making a change using the mouse-based operations is too hard or impossible, the user can still change the output by directly changing the source

code. This approach is made possible thanks to the use of *live synchronization* and *trace-based program synthesis* as defined by Chugh et al. [8]

One example of a programming system that demonstrates this paradigm is called *Sketch-and-Sketch* [7, 8]. It is a browser-based application that provides a bimodal programming environment. It allows creating and modifying Scalable Vector Graphics (SVG) by directly manipulating the program's SVG output. The application consists mainly of an editor window for the source code, a canvas window displaying the program's output, and a button to run the program. After clicking on the canvas, the output is overlaid by context menus and *widgets*, allowing users to modify the program's *sub-expressions* and *intermediate values* as seen in Figure 1.4.

Sub-expressions refer to a certain syntactic scope of the program and the user can select and modify it [8]. *Intermediate values* are values produced at the intermediate execution steps of the program. The program provides widgets for different types of intermediate values such as *offset*, *point*, or *list* widgets. Widgets can take the form of draggable boxes or input fields.

Another programming system implementing the direct manipulation paradigm is *Transformic* introduced by Schreiber et al. [21]. It is a functional interpretation of the Morphic GUI Framework providing direct-manipulation functionality.

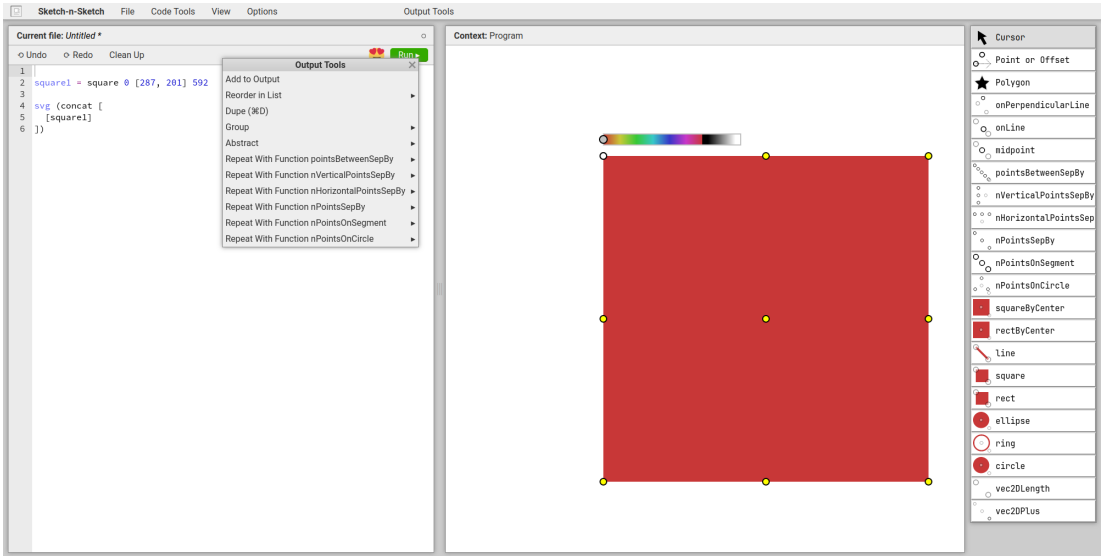

Figure 1.4 Example of Sketch-and-Sketch's user interface. (Image created using the *Sketch-n-Sketch demo application* [22].)

1.2 Web Development

In the previous section, we described the low-code development approach and several existing low-code development systems. In this section, we introduce web development tools and technologies that we use for the *InterfaceSmith* programming system or that influence its design and implementation.

To create a data-driven programming system, we must select a data representation format that is easy for users to obtain, understand, and modify. We choose the *JavaScript Object Notation format* for our input data representation, as it is widely used in web development, is easy to write and understand, and can be easily parsed.

Our programming system provides a GUI, and technologies such as the *Elm* [10] functional programming language influenced the nature of our implementation. We do not use Elm directly, but we model our implementation after the *Elm architecture* [11] architecture, which is a simplified form of the programming model used by Elm.

To implement our programming system, we use the *F#* [23] programming language alongside the *Fable* [24] compiler, which compiles the application into *JavaScript*, which allows the application to run in a browser-based environment.

We use the *Feliz* [25] library to implement the user interface of our programming system. The *Feliz* library provides a *Domain-Specific Language (DSL)* for building *React* user interface components and applications in F#. The *React* [1] library provides tools to create composable components, handle component-level storage and behavior, and handle the re-rendering of specific UI elements.

1.2.1 The JavaScript Object Notation format

The JavaScript Object Notation is defined by Bray [26] as a lightweight, text-based data interchange format. It is language-independent and easy for both humans to read and write and machines to parse and generate. JSON supports the following data types: *Objects*, *Arrays*, *Numbers*, *Strings*, *Booleans*, and the *null* type. We can see an example of the JSON syntax in Figure 1.5

```
{
    "JsonObject" : {},
    "JsonList": [],
    "JsonString" : "Content",
    "JsonNumber": 0,
    "JsonBool" : true,
    "JsonNull": null
}
```

Figure 1.5 Example of the available JSON data types and the JSON syntax

It is widely used in web development for transmitting data between the server and client-side web applications, or used for configuration of different development tools. Thanks to its popularity, many different parsers and other tools have been created to support its use, such as the *Fable.SimpleJson* [27] library which we use in our implementation.

1.2.2 Elm

Elm [10] is a functional programming language presented by Czaplicki and Chong [28]. It allows the creation of responsive graphical user interfaces by employing the *Functional Reactive Programming (FPR)* approach. The functional reactive programming approach applies pure functional programming paradigms to time-varying values, as introduced by Elliott and Hudak [29]. Time-varying values can represent the input and output of the GUI or other information channels, such as server requests. The applications to time-varying values are known as *signals*. Elm simplifies the approach of functional reactive programming, by assuming that the signals are discrete. This means that it avoids unnecessary recomputation when the signals are unchanged. Discrete change of a signal is called an event. Events trigger the recomputation and the application is updated. Elm also provides tools and abstractions to enable asynchronous computations.

The main categories of GUI elements in Elm are *forms* and *elements*. *Elements* represent a *rectangular* GUI element with properties such as width and height. Elements can contain data in a form of text, video or an image and can be composed together. Forms on the other hand represent two dimensional elements of arbitrary shapes with modifiable properties such as color and texture.

Reactive GUIs can be implemented by combining GUI elements and input signals. Elm provides a range of primitive signals such as ”Mouse.clicks” which triggers on mouse click. Some primitive signals may require arguments for their constructor such as the ”Window.dimensions” signal.

1.2.3 The Elm architecture

The *Elm* architecture*Elm architecture* [11], or the *Model View Update(MVU)* architecture, is a programming architecture for building web applications. The architecture is inspired by Elm's FPR programming model but was simplified to make the programming model more usable and easier to learn, as stated by Czaplicki [30].

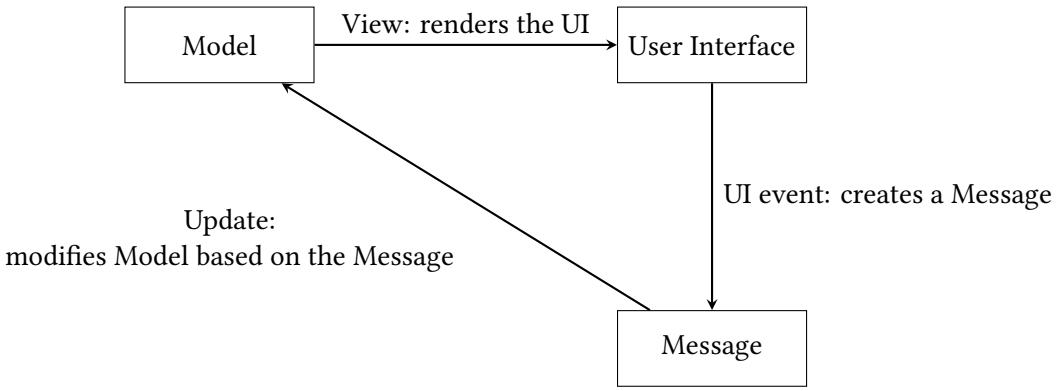

Figure 1.6 Diagram of the Elmish Architecture

In Figure 1.6, we can see a diagram representing the architecture. The architecture consists of the following components:

- **Model:** The *Model* is a data-structure which keeps the state of the entire application. It holds data and other necessary values that the application uses. The data-structure is usually immutable. This means that making changes to the

state results in a creation of an updated copy of the state which becomes the new state.

- **View:** View is a function which takes the current state of the application and creates a graphical user interface. The interface usually consists of multiple GUI elements displaying the application state.
- **Update:** Update is a function which updates the model based on the inputs from the graphical user interface which calls the update function with a certain *Message*. Messages are similar to events in the Elm language. Each message represents a certain change to the model. The update function updates the model based on the provided message and the current application state. The message type is usually represented as a *discriminated union*. This enables the update function to use pattern-matching expressions to efficiently update the model.

1.2.4 F# and Fable

F# [23] is a functional-first programming language running on the .NET platform. Although it is mainly a functional programming language, it allows the use of other paradigms, such as the imperative and object-oriented paradigms. We chose F# to implement our low-code programming system due to its strong type system, functional programming capabilities, and integration with the .NET ecosystem. The reasons for this choice are further elaborated in Chapter 4.

Fable [24] is a compiler capable of compiling programs written in F# to multiple target languages. One of the available target languages is JavaScript, and the compiler produces human-readable and formatted JavaScript code. This enables the creation of web applications in a type-safe functional programming language while also enabling the use of tools and libraries from the wider JavaScript ecosystem.

The Fable compiler supports the use of some types from .NET Base Class and FSharp.Core libraries [31] and attempts to compile these types to native JavaScript whenever possible. This feature allows developers to leverage their existing knowledge of F# and .NET while building web applications.

1.2.5 React and Feliz

The *React* [1] JavaScript library, initially developed by *Facebook*, is used to create user interfaces for various target platforms, such as browsers or mobile devices. The main idea behind this library is to simplify the development process by breaking down complex user interface elements into composable *React components*. Components are the main development units of React applications, and the library provides functionality to implement more complex per-component behavior, such as component-level storage or custom behavior of the UI elements. Components can be implemented as *class-based* or *functional* using *JSX*, which extends the standard JavaScript syntax.

The *Feliz* [25] library provides a DSL to create functional React components in F# Fable projects. It combines the benefits of React, such as the ability to create composable UI elements, and the benefits of F#, such as type safety and functional programming capabilities. We can see an example of a component implemented in both JavaScript and React in Program 1.

Program 1 Comparison of JSX and Feliz syntax

```
//Component implemented using JSX
function SimpleComponent() {
  return (
      <div>
          <p>JavaScript component</p>
      </div>
  );
}

//Component implemented in F# using the Feliz DSL
[<ReactComponent>]
let SimpleComponent =
  Html.div [
      Html.p "Feliz component"
  ]
```

2 Design principles and Features

In the previous chapter, we explored several existing low-code programming system, each one providing different functionality and user interface. In this chapter, we aim to bring together the benefits of the previously described systems, such as the ability to create UI elements based on existing data demonstrated by *AppForge*[9], the responsive development loop of *Darklang* [6], and the ability to generate textual representation of programs based on interactions with a low-code programming interface, as demonstrated by *Sketch-and-Sketch*[7].

Unlike many commercially developed low-code programming systems that focus mainly on supporting specific usage scenarios, we want to approach the design by identifying design principles and following them as closely as possible during our implementation. Our work then explores a particular point in the design space of low-code programming systems. We want to show a minimal but cleanly designed system that satisfies those principles.

We extrapolated three main design principles which we aim to follow during our implementation:

1. **Data-driven UI element creation:** The approach inspired by *AppForge* and *Darklang*.
2. **Low-code editing:** All described systems provide a certain form of low-code editing.
3. **Code generation capability:** The ability to generate text-based representation of the program is mainly inspired by *Sketch-and-Sketch*.

2.1 Data-driven UI element creation

The main general design principle of the proposed programming system is the *data-driven* approach in context of creating web application UI elements. We define the general approach as follows:

Definition 2 (Data-driven approach)**.** *A software development approach where the creation of a particular software artifact is dependent on* ***concrete data****.*

Darklang and *AppForge* are great examples of systems employing this approach as Darklang can create new workers based on existing Cron jobs, or the AppForge system can create a new View based on an existing database table. In the context of creating web application UI elements, some data can be used only for certain UI elements as to maintain the correct semantic meaning and functionality of the elements. Another definition which could be used to describe our approach is the following:

Definition 3 (Data to UI)**.** *Defined by Sahay et al. [4] as "A data-driven approach that starts from data modeling and then builds the user interface of the application followed by the specification of business logic rules and workflows."*

This definition also includes the concept of data modeling but we assume that the data the user provides is already modeled. For example, the developer can use the

proposed system employing this approach together with existing technologies such as *GraphQL* [32]. Using GraphQL, the developer can specify the data they wish to use and its structure. This data can then be provided to the system and used to create the desired web application.

The main difference between our programming system and other systems described by Sahay et al. [4] as *Data to UI* such as *Mendix* [5] is that creating the user interface is directly tied to the data provided to the application. Usually, the *Data to UI* systems provide options to create pre-made UI elements, which may then reference and use the provided data. Whereas our proposed system creates the UI elements based on the data itself which then reference the provided data.

The system analyzes the provided data and provides development options based on its structure and properties. The analysis can be tailored to the specific aims of the programming system. For example, the proposed application for creating web applications can analyze the data and map it to specific HTML elements based on its type and structure. Usually, the developer performs this data analysis and creates the GUI elements by hand. Using this approach, the developer can offload some of this work onto the system and focus on other aspects of the development process.

The data analysis can employ various techniques, such as data structure and data type recognition. One option is that the system performs the analysis once after the developer provides data to the system, and the system provides options based on this single analysis. Alternatively, the system can perform it continuously throughout the development process and provide development options based on the already created application elements, similar to Output-directed programming described in Section 1.1.4. The system could also use various machine-learning techniques to improve the data analysis.

2.2 Low-code editing

The second design principle is the concept of *low-code editing*. Low-code editors usually provide a graphical user interface through which the developer interacts with the system and develops the desired application [3, 4]. The system provides contextual menus to aid in the creation of the application. Various existing low-code programming editors were presented previously in Section 1.1.

The visual design of the *InterfaceSmith* is inspired by *Darklang* and its provided functionality is inspired by the *WYSIWYG* editors as described by Yang et al. [9]. This sub-category of low-code editors displays the application's output without the need to explictly trigger a build step. This design allows the system to update the output every time a change is made to the program and display its result, allowing for a more dynamic development loop [8]. Our prototype system provides contextual menus based on the structure of the input data and places the menus directly next to the preview of the created UI elements, as we will see in the following chapters.

2.3 Code generation

The last design principle is the ability to generate a plain–text representation of the created application. This ability addresses some of the concerns and main pitfalls of many low-code programming systems, as stated by Pinho et al. [3]. These concerns

include *Interoperability issues* and *Vendor lock-in*. Both are mitigated as the user can download the already created part of the application in its textual representation and continue its modification in another programming system.

Our prototype implementation also provides a code generation capability. The type system of the application represents the created parts of the application in a form that allows translation of the created application to various programming languages and other technologies. The prototype system can generate the resulting application in *HTML* and *plain JavaScript*, but could be extended to provide a wider selection of target technologies, such as *React* or *VueJS* frameworks.

3 Data-driven UI Creation

In the previous chapter, we described the design principles that guide the implementation and functionality of the *InterfaceSmith* programming system. This chapter describes the data-driven UI creation feature of the *InterfaceSmith* programming system, expanding upon the principles described in the previous chapter.

Firstly, we describe the *hole-based approach to creating UI elements*, which allows users to create new UI elements based on existing data dynamically. After that, we describe the *domain model*, mainly the internal representation of the UI elements and necessary operations on these types. We then describe mapping the input data types to specific types representing the UI elements. Lastly, we describe how the system combines the input data and UI elements, provides options for creating and modifying the UI elements, and showcase the creation of a simle TO-DO list UI elements. All code examples shown in this chapter are written in the F#[23] programming language.

3.1 Hole-based UI element creation

Our primary motivation is to allow *incremental creation* of the user interface based on concrete data. Rather than creating UI elements for all data fields at once, we provide the functionality to pick what elements to create while still maintaining the structure of the UI based on the input data.

First, the system analyzes the provided data and, based on its structure, creates a UI skeleton containing placeholder types named *holes* that represent potential new UI elements. An alternative explanation is that the *hole* reserves a space for potentially inserting a new element into the UI element tree. Then, the user is provided the option to *fill* these holes with new UI elements that correspond to the particular hole. The creation process of the UI is then a process of gradually filling the holes with new elements.

The elements are created based on the provided data and its structure; each element corresponds to a data field. We define the type named *hole* as follows:

Definition 4 (Hole type definition)**.** *A* hole *is a placeholder type representing a UI element that is yet to be created despite its existing corresponding data. It serves as a temporary stand-in during the incremental construction of a user interface.*

For example, imagine we possess the data shown in Figure 3.1, and we wish to create a web application containing a UI element displaying the value of the *Celsius* field. We upload the data to the programming system, and it presents us with the *hole*

```
{
  "Celsius": 0,
  "Farenheit": 0
}
```

Figure 3.1 Example JSON data

elements for the *Celsius* and *Farenheit* fields. Then, we can choose the option to fill the hole corresponding to the *Celsius* field, which creates a new UI element displaying

the Celsius's value. However, we can also create the UI element for the *Farenheit* field anytime while still maintaining that the UI elements' structure corresponds to the data structure. We can also delete the UI element corresponding to the *Celsius* field, and the system creates the hole again; in other words, we "dig up" the element.

We define the *incremental creation process* as a *sequence of operations*. Each operation is either a *modification* of an existing UI element or a *replacement* of a Hole element with a new UI element based on the corresponding data. This approach allows the system to perform different tasks after each operation. These tasks include updating the UI element preview, analyzing the input data, or providing new modification menus and options.

3.2 Domain model and Mapping

In the previous section, we defined the hole-based UI element creation. To provide this functionality, we must first define the types representing the provided data, the UI elements, and how we create the elements corresponding to specific data fields. In this section, we describe the types that the system uses to represent the input data and the types used to represent the created UI elements, as well as define the mapping between the input data and the UI element representation.

3.2.1 JSON Type

The creation process starts by providing *JSON* data to the system. In order to use this data, we parse it and create an internal representation. We parse the input data using a library called *Fable.SimpleJson* [27]. The library defines a discriminated union type named *JSON*. This type is the foundation for the Abstract Syntax Tree representation of the JSON data. We can see the type definition in Program 2.

Each node represents a JSON value in the input data. The types of nodes can be divided into two categories:

- **Collections:** The first category contains types representing a *collection* of other values. This category includes the types *JObject* and *JArray*. We define the two collection types as follows:
  - **JObject:** It is based on the JSON Object type and represents a collection of different JSON types. The original ordering of the inner elements is *ignored*.
  - **JArray:** It is based on the JSON List type and represents a collection of JSON values of the *same type and structure*. The original ordering of the inner elements is *preserved*.
- **Primitives:** The second category contains types representing data primitives such as numerical values, boolean values, a string of text, or the null value.

Program 2 JSON type

```
type Json =
    | JNumber of float
    | JString of string
    | JBool of bool
    | JNull
    | JArray of Json list
    | JObject of Map<string, Json>
```

3.2.2 RenderingCode Type

To enable the creation of UI elements based on the JSON type, we define a type named *RenderingCode*. The *RenderingCode* is a discriminated union type used to represent the UI elements. Similarly to the JSON type, each case represents a type of an HTML element with a corresponding mapping to the JSON type. This type is the foundation for the *Abstract Syntax Tree* representation of UI elements. We can see the type definition in Program 3.

Program 3 RenderingCode type

```
type RenderingCode =
    | HtmlElement of
        tag: Tag *
        attrs: Attributes *
        innerValue: InnerValue *
        eventHandlers: (string * EventHandler) list
    | HtmlList of
        listType: ListType *
        attrs: Attributes *
        itemCodes: RenderingCode list *
        eventHandlers: (string * EventHandler) list
    | HtmlObject of
        objectType: ObjType *
        attrs: Attributes *
        keyOrdering: string list *
        codes: Map<string, RenderingCode> *
        eventHandlers: (string * EventHandler) list
    | Hole of FieldHole
```

Attributes

The RenderingCode type, except the case type *Hole*, provides the ability to create and modify custom *Attributes*. The *Attributes* type represents a collection of key-value pairs representing HTML attributes. An *Attribute* consists of a *key* of a type *string* and an *value* of a type *InnerValue* whose definition we see in Program 6. We can see the type definitions for the *Attributes* and *Attribute* types in Program 4.

Program 4 Attribute and Attributes type definition

```
type Attribute = {
    Key: string
    Value: InnerValue
    Namespace: string option
}

and Attributes = Attribute list
```

Each HTML element can contain several associated attributes, and different types of HTML elements allow the use of unique attributes, such as the *type* attribute for the *input* HTML element. Our editor allows the definition and modification of these attributes through a low-code interface we see in Figure 3.2. The user can create an attribute of a particular key and modify the value by selecting the InnerValue case type they wish to use.

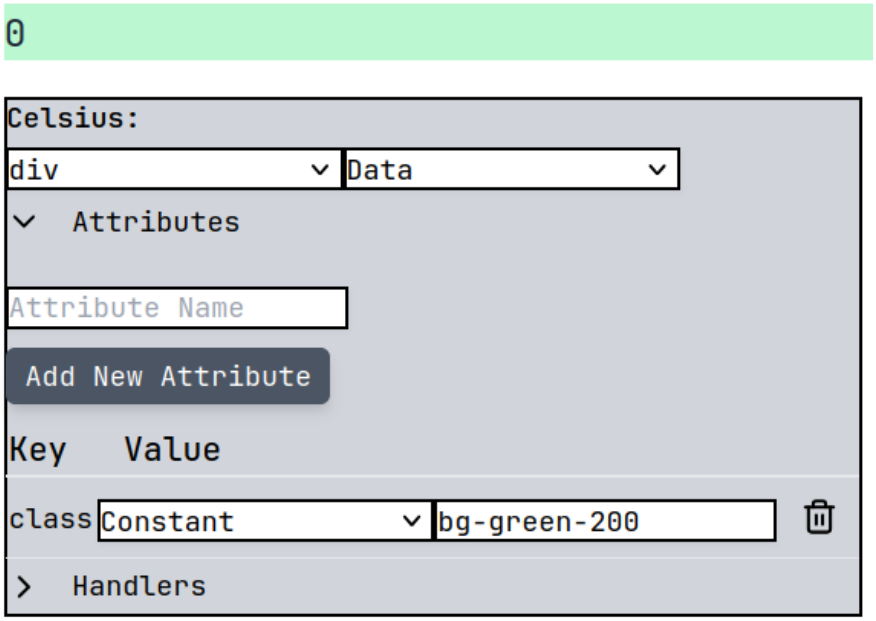

Figure 3.2 Attribute modification menu for a HtmlElement

Event Handler

Each RenderingCode type, except the case type *Hole*, also provides the ability to add custom *event handling* functionality. A *handler* is a function called when a certain HTML *event* occurs, such as when the element is hovered over or clicked. Each *EventHandler* consists of the name of a *function* or an *elmish-style message*, depending on how the user wishes to implement custom element behavior, and the definition is displayed in Program 5.

Program 5 EventHandler type definition

```
type EventHandler =
    | JsHandler of functionName: string
    | MsgHandler of the message: string
```

We see the menu for assigning the event handling functionality to UI elements in Figure 3.3. It provides options to select an *event* and assign a corresponding handler and also indicates whether the selected handler is a JavaScript function or an Elm-style message.

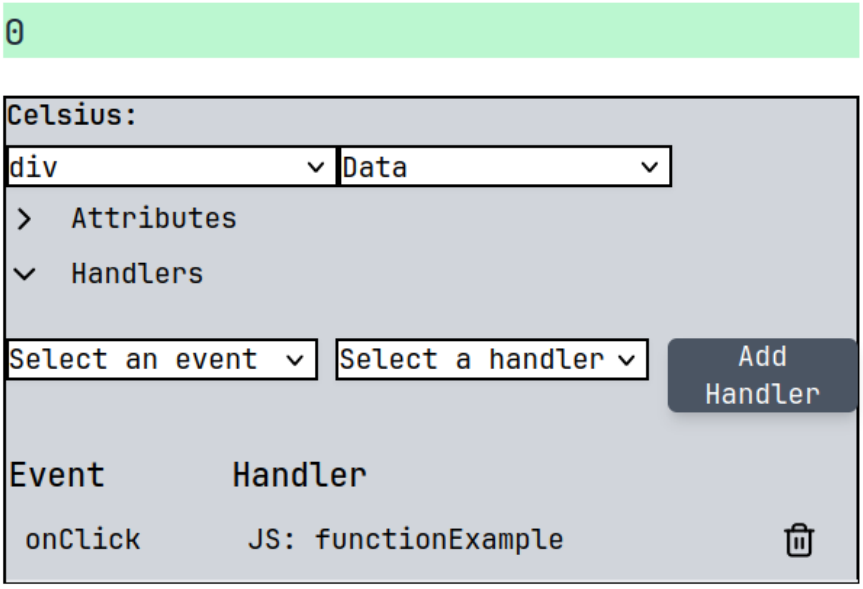

Figure 3.3 EventHandler modification menu for a HtmlElement

Hole

The *Hole* type represents a placeholder in the UI element structure as defined in Section 3.1, allowing for dynamic element creation. There are two types of holes, *named* and *unnamed*. A named hole corresponds to a specific data field and bears its name. An unnamed hole corresponds to a field with the *null* value, for which the system cannot create a new UI element.

HtmlElement

The *HtmlElement* type represents a single HTML element. It consists of a tag of a type *Tag*, which denotes the HTML tag (e.g., div, p, span), the *InnerValue* represents the content of the element defined as seen in Program 6, and the previously defined attributes and event handlers.

The InnerValue's *Data* case type indicates content from the corresponding input JSON value, *Empty* represents an empty element, and *Constant* holds static, developer-defined content.

Program 6 InnerValue type definition

```
type InnerValue =
    | Data
    | Constant of string
    | Empty
```

HtmlList

The *HtmlList* type represents a collection of ordered HTML elements of the *same type and structure* corresponding to JSON arrays. The first type it consists of is the *listType*, which specifies the list type (e.g., unordered, ordered), and we define it as a simple discriminated union type. Next, the *itemCodes* of a type *RenderingCode list* is a collection of RenderingCode elements representing the list's contents. Lastly, it also consists of the previously defined attributes and event handlers. As we assume that the list consists of elements of the same type and structure, when one list element is modified, we can apply these modifications to all remaining elements.

HtmlObject

The *HtmlObject* type represents an unordered collection of *diverse* HTML elements derived from a JSON object. Firstly, it consists of an *objectType*, which defines the object type (e.g., div, span, section) and is defined as a simple discriminated union. Next, the *elements*, which is a list of RenderingCode elements representing the object's contents, *keyOrdering*, which defines the custom rendering order for the fields, and the previously defined *attributes* and *event handlers*.

3.2.3 Data Mapping: JSON to RenderingCode

Transforming JSON data into UI elements requires systematic mapping between JSON and RenderingCode types. This mapping forms the core of our UI generation system, allowing us to convert arbitrary JSON input into a structured representation of UI elements. Each JSON type has a corresponding representation in RenderingCode:

- JObject maps to HtmlObject
- JArray maps to HtmlList
- JNull, JString, JNumber, and JBool map to HtmlElement

The *incremental creation process* described in Section 3.1 involves dynamically creating replaceable *Hole* elements. We perform this creation lazily and incorporate it into the mapping process. A *Hole* element is created for each inner element of the collection types described in Section 3.2.1.

We call this mapping process the *recognition* of the JSON value and define the mapping function named *recognizeJson* in Program 7.

Program 7 JSON to RenderingCode mapping

```
let recognizeJson (json) =
    match json with
    | JArray array ->
        //create a hole for each element of the array
        //return a HtmlList containing the holes as elements
    | JObject obj ->
        //create a Hole for each element of the object
        //return a HtmlObject containing the holes as elements
    | JNull -> Hole(UnNamed)
    | _ -> HtmlElement
```

3.3 Incremental Creation Process

We described the *incremental creation process* briefly in Section 3.1 as a sequence of operations. In this Section, we extend this description and define the previously described operations in greater detail. We divide the operations into two categories:

- **User operations:** Operations performed by the user such as element creation, element modification, and providing data to the system. User operations are performed through the provided GUI.
- **System operations:** Operations performed by the system in reaction to the User operations. These operations include creating new RenderingCode elements, analyzing newly visited JSON values, modifying AST, rendering menus, and previewing elements.

This categorization allows us to divide the functionality and responsibilities of the system between different parts of the application. For example, the GUI portion of the application is mainly responsible for accepting User operations, whereas other modules can implement different System operations.

3.3.1 Hole Replacement and Modification

User operations require finding a specific node inside the RenderingCode AST during creation. To find this specific node, we define a dynamically generated *Path* for all nodes during the traversal process. The *Path* is a sequence of indices unique to every node in the AST.

Program 8 Example RenderingCode AST with corresponding paths

```
  HtmlObject(ObjType.Div, [],["key1";"key2"], [ //Path: []
      HtmlElement(Tags.div,[], InnerValue.Empty, []) //Path: [0]
      HtmlList(ListType.UnorderedList,[], [ //Path: [1]
          HtmlElement(Li, [], Constant "Item 1", []) //Path: [1,0]
          HtmlElement(Li, [], Constant "Item 2", []) //Path: [1,1]
        ], [])
    ], [])
```

We can see the different paths for each element in Program 8, which shows an example of RenderingCode AST. The example AST consists of a root node of a type HtmlObject and elements contained within it. We see that the root node has an empty path. Then, we append the position of each element inside the collection to the path. This allows us to traverse the AST based on the specified path and find the correct element using a recursive search function.

Using the dynamically created paths, we can replace a RenderingCode by following its specified path through the RenderingCode AST. We define a recursive function named *replace*, which navigates the AST using the provided path to locate and replace a specific element with a new one. We can see a description of the potential implementation in Program 9.

The User operations such as replacing a Hole with a new RenderingCode or modification of an existing RenderingCode correspond to a specific usage of the replace function.

Program 9 A function used to replace a RenderingCode inside the RenderingCode AST

```
let rec replace
  (path: int list)
  (replacementCode: RenderingCode)
  (currentCode: RenderingCode) =
  match path with
  | [] -> replacementElement
  | head :: tail ->
      match currentCode with
      | HtmlList ->
          // recursively call the function on a collection element
             with the index equal to head
          // return new HtmlList
      | HtmlObject ->
          // recursively call the function on an element with
          // the key located inside the keys collection, that has
             index equal to head
          // return new HtmlObject
      | _ -> currentCode // remaining path, but the current element
         is not a collection -> no changes to the RenderingCode
```

3.3.2 Traversal

An essential aspect of the creation process is how the system processes the JSON AST and helps users build the RenderingCode AST. The process involves traversing both the JSON and RenderingCode ASTs simultaneously. Our system can perform this simultaneous traversal because we create the RenderingCode AST based on the structure of the existing JSON AST. This allows the system to dynamically create Hole elements for direct descendants of existing RenderingCode elements for which corresponding data exists, but the elements have not yet been created.

Another way to look at this process is that the RenderingCode AST is guiding our traversal, while the JSON AST inspires the structure of the RenderingCode AST. The traversal begins from the roots of the ASTs, and we traverse the trees recursively. Based on the type of the visited RenderingCode node and the mapping described in Section 3.2.3, we can perform different operations:

- **HtmlElement:** The node is of type HtmlElement, which maps to a primitive JSON value. This means the JSON value has no descendants, and we can end the traversal. The system then performs the operations of displaying a preview of the HtmlElement and a modification menu. We can see an exammple of the menu and the preview of the HtmlElement displayed in Figure 3.4

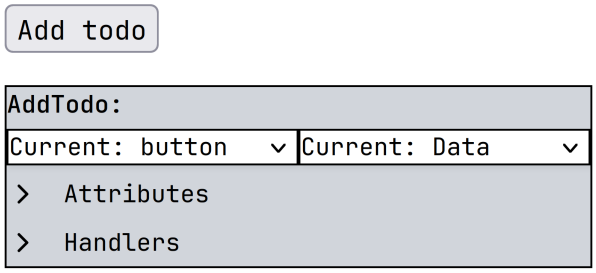

Figure 3.4 A modification menu and a preview for a HtmlList element.

- **Hole:** The node is of type Hole, meaning the JSON value was visited before, but the user has not created the corresponding RenderingCode. The presence of a Hole element also means that we cannot visit the potential descendants, and we can stop the traversal. The system displays a menu to add a new element corresponding to the JSON value. We see the menu displayed in Figure 3.5.

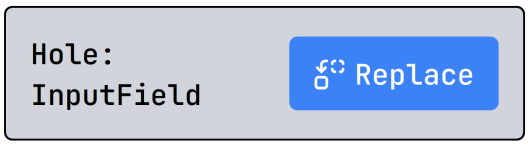

Figure 3.5 The menu used to replace a *Hole* element.

- **HtmlList and HtmlObject:** When the node represents a *Collection* type of type HtmlList or HtmlObject, we recursively traverse all its descendants. The system displays a modification menu for the element and continues the traversal.We see the modification menu for the HtmlList displayed in Figure 3.6 and the modification menu for the HtmlObject in Figure 3.7.

After every User operation, the system performs this traversal to update the element preview and modification menus to reflect changes made to the RenderingCode AST.

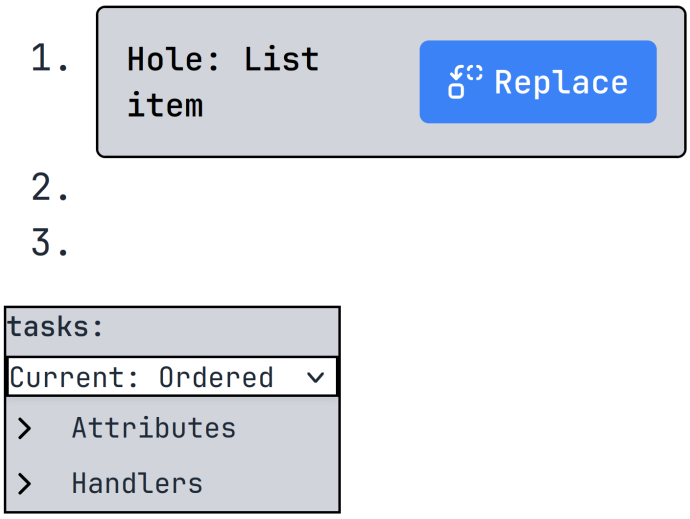

Figure 3.6 A modification menu for HtmlList element.

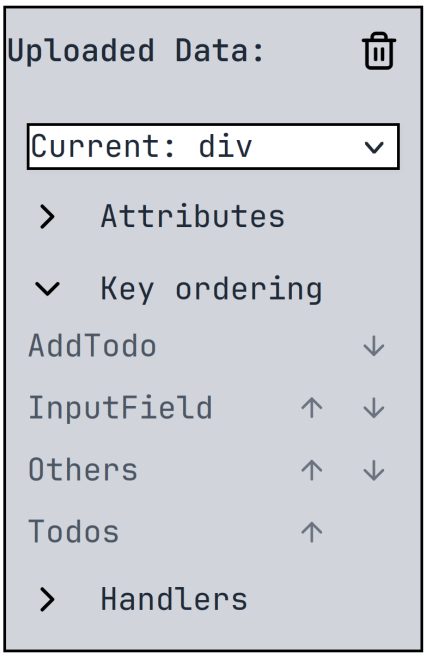

Figure 3.7 A modification menu for the HtmlObject.

3.3.3 Combining RenderingCode and JSON data

Once the RenderingCode AST is created, the system must combine the RenderingCode elements with the data from the JSON AST. We use *Structural referencing* to combine a RenderingCode element with the corresponding JSON value. Structural referencing describes a process where a RenderingCode with a specific assigned Path is combined with a JSON value with the same Path. We can do this easily, as the structure of the RenderingCode AST mirrors that of the JSON AST. Thanks to the approach of Structural referencing, we can perform this process during our previously described traversal in Section 3.3.2.

3.3.4 Creation of a single RenderingCode

Creating a single RenderingCode element consists of multiple System and User operations. We can see the sequence of operations in Figure 3.8.

The process starts when the system visits a previously unvisited node of the JSON AST during the traversal described in Section 3.3.2. The system automatically creates a corresponding Hole element and adds it to the RenderingCode AST. Then, the system displays a menu to the user, allowing them to replace the Hole element with a RenderingCode based on the JSON value. After the user chooses to add the new RenderingCode element, the system uses the previously defined *recognizeJson* function to create the new element based on the corresponding JSON value. Following that, the system uses the *replace* function to replace the existing Hole element at the specified path with the newly created RenderingCode. Lastly, the system traverses the modified RenderingCode AST and JSON AST described in Section 3.3.2 and performs operations, such as displaying modification menus and element preview.

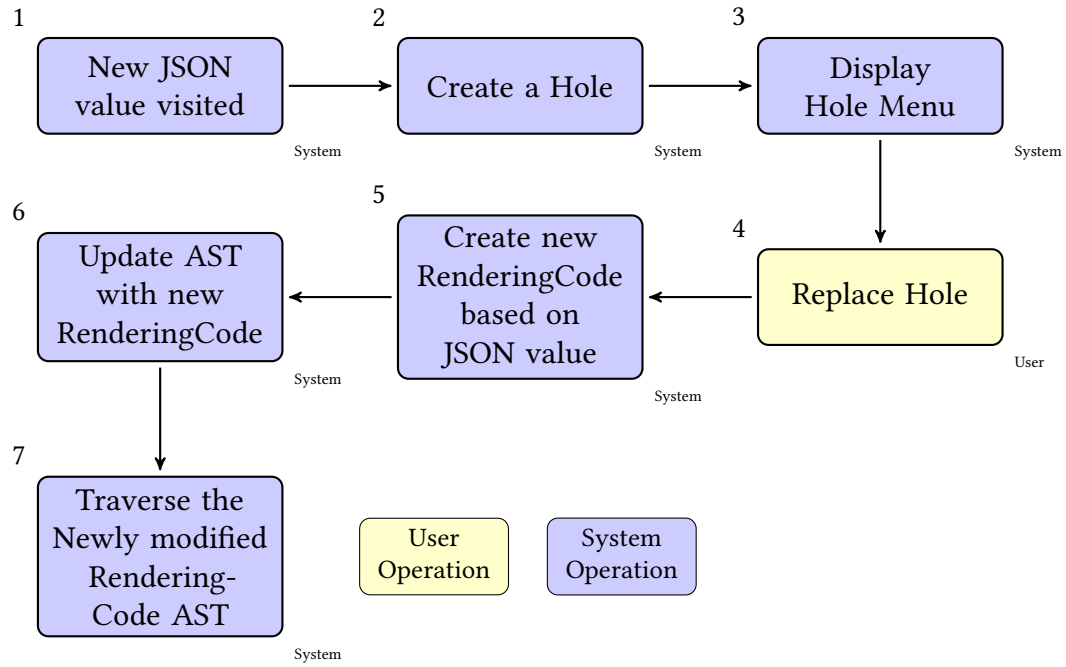

Figure 3.8 Single RenderingCode Creation Process

3.4 Simple TODO list UI creation

In the previous sections, we described the domain and operations necessary to provide the *data-driven incremental UI creation* functionality. To help visualize the process of creating a UI based on concrete values, we create a simple UI for a TODO list application based on JSON data displayed in Figure 3.9. We see the steps of the entire process in Figure 3.10.

```
{
  "title": "TODO list",
  "tasks": [
      { "task": "Complete project proposal"},
      { "task": "Prepare presentation slides"},
      { "task": "Send meeting agenda to team"}
  ]
}
```

Figure 3.9 Example TODO list JSON data

The process starts by uploading the JSON data displayed in Figure 3.9 to the system. The system analyzes it and creates a UI skeleton displaying the hole menus for the *title* and *tasks* fields.

We then click on the button to replace the *hole* corresponding to the title field. The system automatically creates a new HtmlElement, displaying its preview and a modification menu. We repeat the process for the tasks field, and the system creates a new HtmlElement and a hole for the list elements.

As described in the previous section, we assume the list contains elements of the same type and structure, allowing the system to show modification menus only for the first list element. All changes made to it affect all other elements. After replacing the modification hole, the system finally creates the final HTML elements for each list element, displaying modification menus for all RenderingCode elements. Finally, we click the *Toggle Options* button and see the preview of the created elements without the modification menus present.

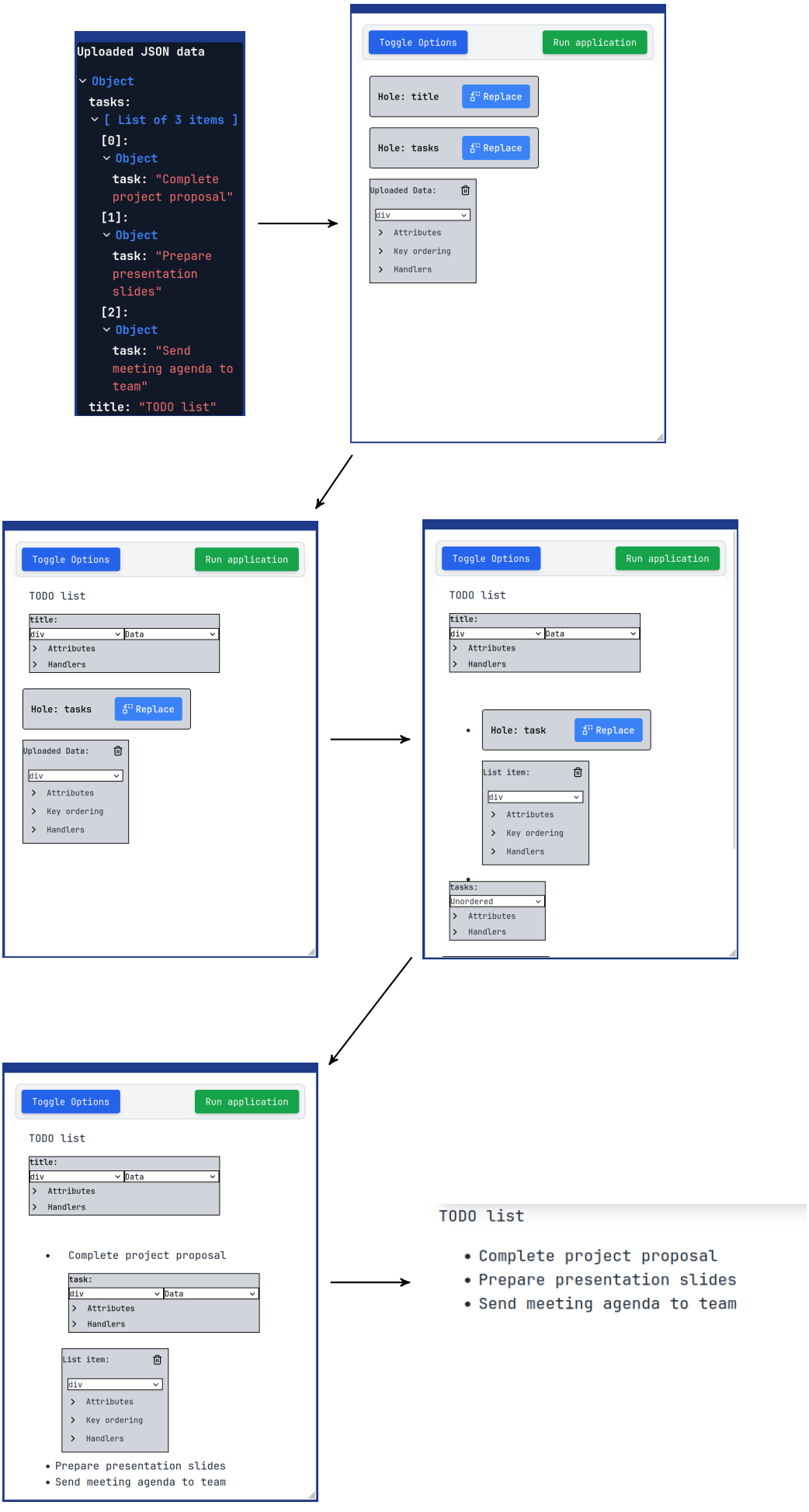

Figure 3.10 Step-by-step creation of a TODO list from JSON data.

3.5 Summary

This chapter describes our approach to creating UI elements based on concrete data. We defined the hole-based approach for incrementally creating UI elements, the domain we use to represent the created UI elements and data, the operations on these types, and how the created elements utilize the input data. We concluded the chapter by showing the functionality of the resulting system on an example creation of a simple TODO list UI.

- In Section 3.1, we defined the placeholder type named *hole* and the hole-based approach for incrementally creating UI elements based on concrete data.
- In Section 3.2, we defined the application's domain model consisting of types representing the provided data and UI elements. We described the representation of JSON values provided by the *Fable.SimpleJson* [27] library and the representation of the created UI elements using our custom *RenderingCode* type. We also described the mapping between the input data and our internal representation of created UI elements.
- In Section 3.3, we described the *Incremental Creation Process* including the User and System operation. Then, we described replacing and modifying the RenderingCode AST using a Path-based approach. After that, we described the simultaneous traversal of the JSON AST and RenderingCode AST. Lastly, we described combining the RenderingCode elements with corresponding JSON values.
- In Section 3.4, we showcased the creation of a simple TODO list user interface using the *InterfaceSmith* system, and the process is displayed in Figure 3.10.

4 Implementation

In the previous chapter, we described the theoretical background for the *InterfaceSmith* programming system, defined the types representing the input data and UI elements, described operations on these types, and showcased the system's core functionality. In this chapter, we describe the implementation according to the design principles described in Chapter 2 and the data-driven UI creation described in Chapter 3. We explore the technologies chosen, the system's architecture, and key implementation details.

4.1 Technologies

To provide context for the implementation, we'll first examine the technologies used in developing the *InterfaceSmith* system.

We created the *InterfaceSmith* programming system as a browser-based client-side application providing a graphical user interface rather than a traditional desktop application. The main reasons were to allow cross-platform compatibility and the ability to see the preview of the created web applications in a browser-based environment.

To implement the application, we employed various existing tools that aligned with our vision for the application and fulfilled our demands on the functionality they provide. We wanted to implement the system using a programming language with functional programming capabilities and a strong ecosystem, which narrowed our range of suitable options and subsequently influenced which supplementary technologies we could use. The system is implemented using the following key technologies:

- **F#:** The *F#* [23] programming language is used to implement the entire application, including the core logic and the user interface, chosen for its strong type system and functional programming capabilities.
- **Fable:** The *Fable* [24] compiler, briefly described in Section 1.2.4, compiles the F# source code to JavaScript, enabling browser-based execution and using technologies from the JavaScript ecosystem.
- **React:** The *Feliz* [25] library provides a domain-specific language (DSL) for building *React* user interface components and applications in F#.
- **Elmish:** *Elmish* [33] is a library used to enable the creation of Elmish style applications in F#, which follow the MVU pattern described in Section 1.2.3.
- **Tailwind:** We use the Tailwind CSS framework for the layout and styling of the UI components of the application, which provides composable CSS classes and enables high customizability of the UI elements.
- **SimpleJson:** The *Fable.SimpleJson* [27] library is used to parse the input JSON data into the internal representation described in Section 3.2.1.
- **SAFE stack template:** We use the *SAFE stack* [34] template's *Build* project, which provides scripts for building web applications built in F#.

4.1.1 Alternatives

While we selected the technologies mentioned in the previous Section for our implementation, for several of them, we considered using alternative technologies:

- **Programming Language:** Instead of F#, we could have used other programming languages such as Haskell, OCaml, or TypeScript, which also offer strong type systems and functional programming capabilities. However, F# was chosen for its seamless integration with the .NET ecosystem, high-quality development tools and documentation, and its ability to be compiled to JavaScript via Fable.
- **CSS Framework:** We considered using alternative CSS frameworks, such as Bulma or Bootstrap, which provide pre-made styled components. We chose Tailwind instead, as the composable styling classes allow for a more direct approach to styling UI elements instead of trying to adapt and modify the pre-made components.
- **JSON Parsing:** Instead of Fable.SimpleJson, we could have used the closest alternative library called *Thoth.Json* [35]. The main strength of this library is the ability to create custom JSON *encoders* and *decoders*. However, as we need the ability to parse JSON data of arbitrary structure, SimpleJson's lightweight nature and its internal representation of the parsed data made it our preferred choice.

While these alternatives have their strengths, our chosen technologies provided the best balance of functional programming capabilities, browser compatibility, and ecosystem support for our specific requirements.

4.2 System architecture

Before we describe the implementation specifics, we must first describe the *architecture* of the implementation. As the main goal of the system is to allow users to create web applications based on concrete value, we decided that our main implementation unit will be a *Page*, which comprises data, UI elements and custom functionality. We see the definition of the *Page* in Program 10.

The application follows a nested Elmish architecture approach. The main application is implemented as an Elmish application responsible for the general functionality, such as Page creation, deletion, and state management across all pages. Within this main application, each individual Page is managed by its own Elmish *PageEditor* sub-application. This provides clear separation of concerns, as the main application focuses on the management of all Pages and holds the state for all of the PageEditor sub-applications, each PageEditor sub-application independently handles the modification of its individual Page.

Program 10 The Page type definition

```
type Page = {
    Name: string
    Id: Guid
    ParsedJson: Json
    CurrentTree: RenderingCode
    JsonString: string
    UserMessages: UserMessage list
    UpdateFunction: UpdateFunction
    CustomFunctions: Map<string, Javascript>
}
```

Main Elmish application

At the top level, our application follows the Elm architecture described in Section 1.2.3. The primary application state is represented by the *Model* type, and we define corresponding *view* and *update* functions to manage this state.

We can see the *Model* type representing the entire main application's state in Program 11. It consists of a collection of created pages, collection to store the ordering of the pages based on when they are created, the id of the currently open Page, and a variable of wheter the sidebar menu is open.

The *Update* function updates the state based on a *Message* it recieves and we see the *Msg* type defined in Program 12. The messages represent events such as creating a new page or toggling the sidebar menu, or updating the state of a certain Page Editor.

The *View* function renders the main general application elements, aswell as the Page Editor application for the selected page. We use the *Feliz* [25] library to create *React* components and style them using *Tailwind* [36].

Program 11 Definition of the Main Elmish application's Model.

```
type Model = {
    Pages: Map<Guid, PageEditorModel>
    PageOrder: Guid list
    ActivePageId: Guid option
    IsSidebarOpen: bool
}
```

Program 12 Definition of the Main Elmish application's Msg type.

```
type Msg =
    | CreatePage
    | UpdatePage of PageEditorModel
    | DeletePage of Guid
    | ToggleSidebar
    | OpenPage of Guid
    | PageEditorMsg of Guid * PageEditorMsg
```

PageEditor sub-application

The *PageEditor* is an Elmish application that provides modification functionality for an individual Page. It is separate from the Main application and has its own local state, update logic, and view functions while still being nested in the Main application. It provides the functionality described in Chapter 2 and Chapter 3. The state of the application is represented by the *PageEditorModel*, and we define a corresponding *PageEditorView* React component and the *pageEditorUpdate* function to manage its state.

The *PageEditorView* is implemented as a React functional component using the Feliz DSL. The UI is inspired by the *Darklang* [6] programming system and features a movable canvas containing draggable elements. We describe the UI elements in greater detail in the following sections.

We see the *PageEditorModel* representing the state of the PageEditor application in Program 13. It consists of the specific *Page*, types for rendering the UI elements of the editor, and types for rendering the movable canvas and its elements.

The *pageEditorUpdate* function updates the state based on the *PageEditorMsg* dispatched by the *PageEditorView*. The main categories of the PageEditor messages are:

- Input/Output operations, such as handling file upload and download events.
- Interactions with the PageEditor UI that do not change the state of the *Page*.
- Modification the Page's RenderingCode AST.
- Modification of the Page's custom functions.
- Modification of the Page's Elmish-style messages.

Program 13 The PageEditorModel type representing the state a PageEditor application.

```
type PageEditorModel = {
    PageData: Page
    FileUploadError: FileValidationError option
    ViewportPosition: Position
    Scale: float
    Elements: Element list
    DraggingElementId: int option
    IsPanning: bool
    LastMousePosition: Position option
    IsPreviewOpen: bool
    IsCodeViewOpen: bool
}
```

4.2.1 Module structure

The implementation is divided between different F# *modules*, each providing different functionality. We define two main modules named *Core Logic* and *Editor*, which contain sub-modules implementing specific functionality. These modules can be described as follows:

1. **Core Logic module:** Contains modules responsible for implementing the Core logic described in Chapter 3, such as the representation of the UI elements and the various operations on these types.

2. **Editor module:** Modules focused on implementing the user interface of the programming system and its functionality following the Elm architecture described in Section 1.2.3. The modules use and depend on the functionality provided by the CoreLogic module.

Each sub-module comprises various functions or type definitions. In our implementation, we separate type definitions and functions into distinct modules, similarly to the separation of *Domain* and *Infrastructure* layers of the *Domain-driven* architecture. The modules containing the type definitions define the system's domain and have either none or a small number of external dependencies. The modules containing the functions implement the behaviors and operations that manipulate and utilize the domain types. This approach inherently makes the application more easily extensible. For example, we can add new functionality to the Editor module without changing the domain model or the implementation of the Core Logic submodules.

Core Logic Module

The Core Logic module is the first main module of the *InterfaceSmith's* implementation. It contains the implementation of the types and operations described in Chapter 3. Figure 4.1 illustrates the overall structure of this module. We divide the implementation into the following sub-modules:

- **Types:** This module comprises sub-modules that define the fundamental data structures of our system:
  - **RenderingTypes:** Contains definitions for types such as the *RenderingCode*, *InnerValue*, and other related types.
- **Operations:** This module includes sub-modules that implement various operations on the types defined in the Types module:
  - **RenderingCode:** Implements operations such as the *replace* function, which modifies the RenderingCode AST.
  - **DataRecognition:** Handles the mapping process between input data and our internal UI element representation. It includes the *recognizeJson* function, detailed in Section 3.2.3.
  - **CodeGeneration:** Provides functionality to generate textual representations of created web applications from RenderingCode ASTs. Our implementation generates an MVU-style application in pure JavaScript, which we describe in the following sections.

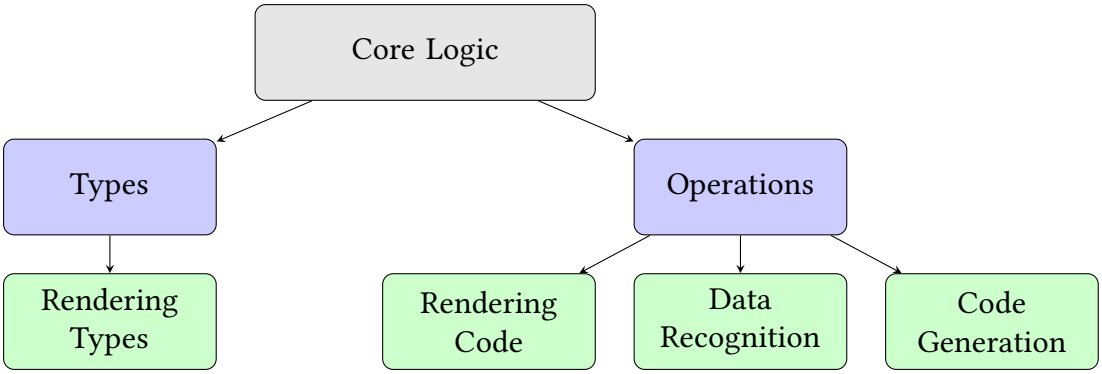

Figure 4.1 Core Logic Module Structure

Editor module

The Editor module is the second main module of the *InterfaceSmith's* implementation. It is responsible for implementing the programming system's UI, state management, and functionality.

As our main implementation unit, we selected a Figure 4.2 illustrates the module's structure, and we describe the implementation in greater detail in the following sections. We divide the module into the following sub-modules:

- **Types:** This module defines the core data structures that represent the state and operations of our Elmish applications, as detailed in Section 1.2.3.
  - **EditorDomain:** Definition of the domain types that represent the internal state of the entire application.
  - **PageEditorDomain:** Defines specialized domain types focused on representing the *PageEditor* Elmish application.
- **Utilities:** Provides a collection of helper functions that support various aspects of the application:
  - **Icons:** Icon importing and management.
  - **FileUtilities:** File IO operations.
  - **JsonParsing:** JSON data parsing and serialization, leveraging the *Fable.SimpleJson* [27] library.
  - **JavaScriptEditor:** Importing the libraries necessary to use the Codemirror editor component.
- **Components:** Implements custom components that form the interactive user interface of the Page Editor sub-applications:
  - **ElementComponents:** Implementation of the draggable canvas elements.
  - **OptionsComponents:** Provides components serving as context menus to allow editing functionality for the individual RenderingCode elements.
  - **EditorComponents:** Provides a set of components that deliver general editor functionality, such as a collapsable side menu showing the created pages.

  - **PageEditorComponents:** Implements the *PageEditor* Elmish application used for editing a specific *Page*.

- **CustomRendering:** Implements dynamically rendering previews of the RenderingCode AST, together with interactive modification menus, and the rendering of the canvas elements.

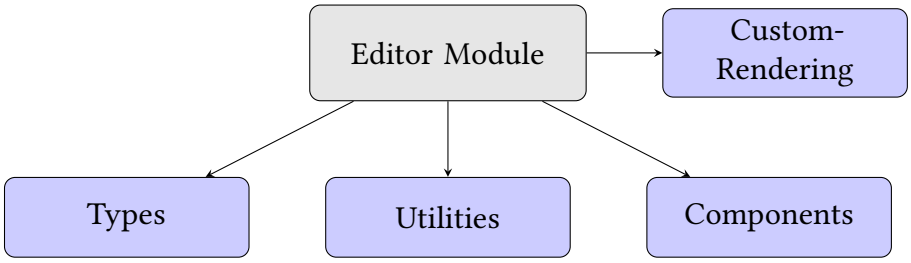

Figure 4.2 Editor Module Structure

4.3 Main Features

Now that we have defined the implementation's modular architecture and structure, we can describe the specifics of the implementation. As the implementation comprises many different type definitions and functions of various complexity, we will briefly describe the system's *main features* that implement the design principles described in Chapter 2.

To implement the programming system, we use tools and technologies previously described in Section 4.1. One of these tools is the *Feliz* [25] library, used for creating composable *React* UI components in F#. Another important tool is the *Elmish* [33] library, which provides abstractions that enable the creation of *Elm-style* applications in F#.

4.3.1 User Interface implementation

The integral component of our programming system is the low-code user interface, as it realizes the design principles described in Chapter 2. It provides tools for creation and modification of UI elements through the use of context menus, shows the preview of the already created UI elements, and allows customization of the functionality of the created UI elements. The user interface is composed of multiple React components some managed by the top-level Elmish application, while others managed by the individual PageEditor sub-applications.

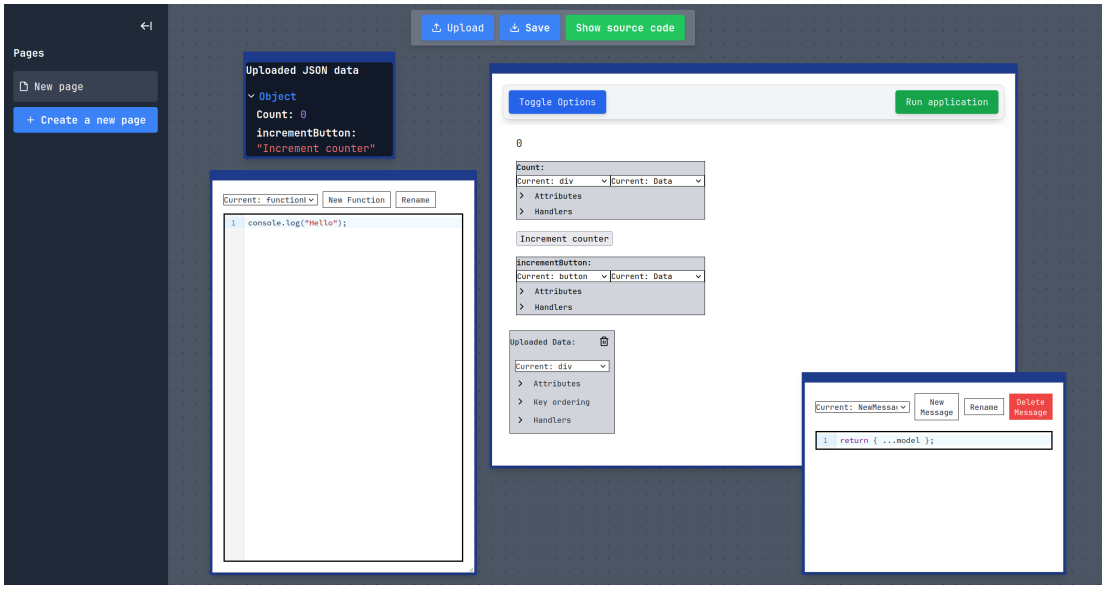

Figure 4.3 Example of *Data-driven UI's* user interface

We see the layout of the *Data-driven UI's* interface in Figure 4.3. Located on the left is a collapsible side-menu used to open and delete pages, and a button to create a new page. Users can also customize the name of the individual page, by clicking twice on the page option. The menu is implemented as a *React component*, and dispatches different messages to the *update* function based on the interactions with the UI. The state and operations of this menu is managed by the main Elmish application.

To the right side of the menu we see the movable canvas, displaying canvas elements for the selected page. On the top of the canvas we can see a panel with three buttons, the first one is used to upload the input data for the page, the second one is used to download the generated JavaScript source code for the created page, and the third button is used to preview the generated JavaScript code for the page. The canvas elements are draggable and some are also resizable. The state and operations of the canvas and its content is managed by the individual PageEditor applications.

Canvas elements

- **ModelElement:** Displays the uploaded data. Each field is collapsible and also shows the type of the field. We an example of this element in Figure 4.4.
- **FunctionsElement:** Provides options to create, delete and modify custom functions. It also provides an editor window to write the function implementation. We see the element displayed in Figure 4.5.
- **MessageAndUpdateElement:** Allows the user to create new messages and modify the update function's modification of the model.
- **ViewElement:** A resizable element displaying a preview of the created UI elements alongside modification menus described in Chapter3 for each element. Also serves as a preview window of the running application.

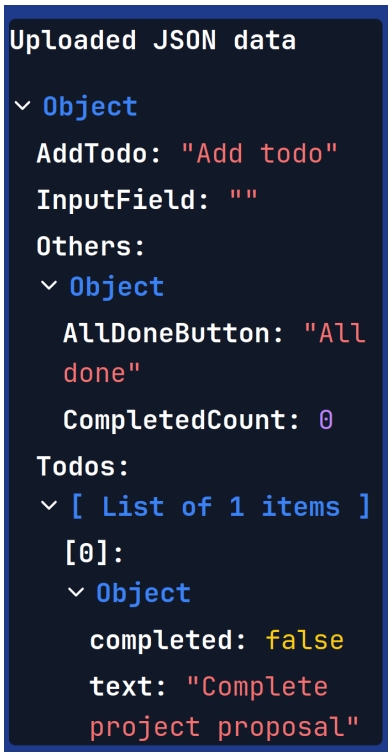

Figure 4.4 The canvas ModelElement showing a preview of the uploaded JSON data.

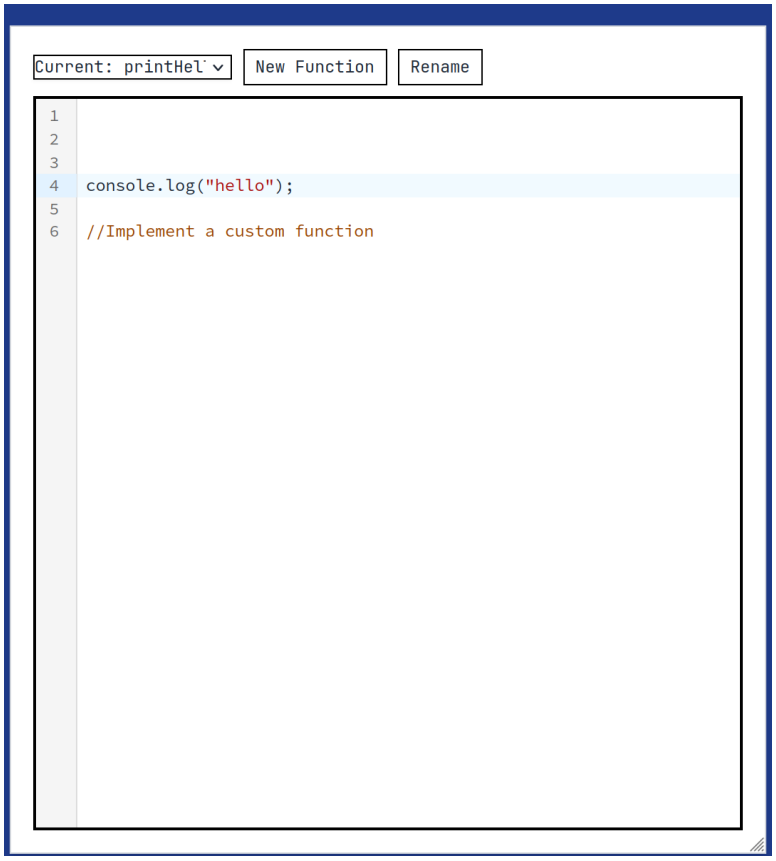

Figure 4.5 The canvas FunctionsElement

Figure 4.6 The canvas MessageAndUpdateElement.

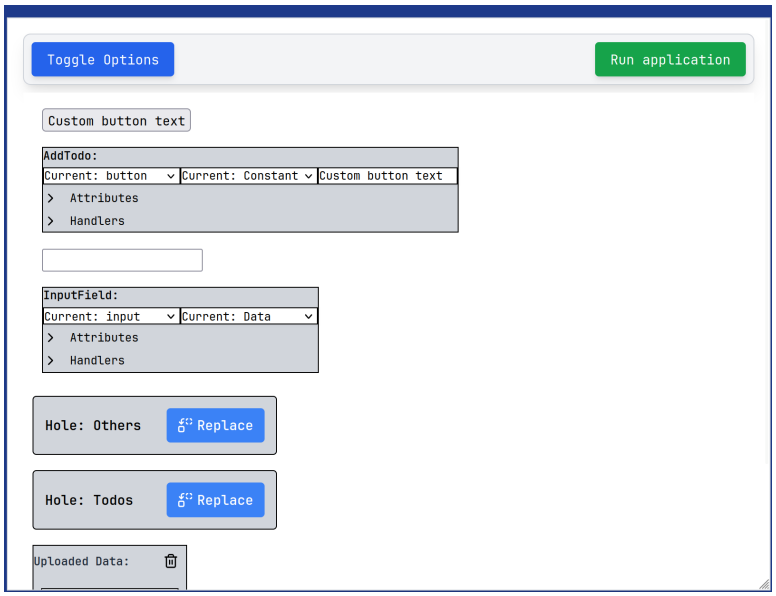

Figure 4.7 The canvas ViewElement.

4.3.2 Custom Rendering

To implement the Low-code approach described in Chapter 2 combined with the Data-driven approach described in Chapter 3, we define a *Higher-order* recursive function named *renderingCodeToReactElement*. This function performs the simultaneous traversal of the *RenderingCode* and JSON ASTs described in Chapter 3. This function provides the main functionality of our implementation and effectively combines the functionality from the *Core logic* module, with the low-code approach. It is used by the previously described *ViewElement*.

The function recursively traverses the RenderingCode AST, matching each type of RenderingCode to its appropriate rendering function. We then define a rendering function for each type of RenderingCode. Each rendering function combines the RenderingCode structure with the corresponding JSON data to create a preview of the element. The function also interweaves option menus with the rendered elements, allowing for editing of the rendered structure. Error handling is implemented throughout the rendering process to handle mismatches between the RenderingCode and JSON structures.

The function takes two arguments, the current RenderingCode node and *RenderingContext*. The RenderingContext type provides information necessary for correctly rendering the preview of the RenderingCode and the modification menu. We define the RenderingContext type as seen in Program 14.

Program 14 The RenderingContext type definition.

```
type RenderContext<'Msg> = {
    Options:
        ('Msg -> unit)
            -> RenderingCode
            -> list<int>
            -> string
            -> Map<string, Javascript>
            -> UserMessage list
            -> ReactElement
    Dispatch: 'Msg -> unit
    Json: Json
    Path: int list
    Name: string
    CustomFunctions: Map<string, Javascript>
    ShowOptions: bool
    UserMessages: UserMessage list
}
```

4.3.3 Code generation

One of our core design principles, as outlined in Chapter 2, is the ability to generate a textual representation of the created application. We define a recursive function named *generateJavaScript* to implement this functionality. It generates a complete JavaScript application using the *Elm architecture* [11], transforming RenderingCode into client-side JavaScript code.

As the user uploads the JSON data in its text representation to the application, we can directly use it. We create the *Model* type, equal to the uploaded JSON and representing the application's state. This then allows us to reference this model and also change the state.

The *view* functions are generated using a recursive function called *generateView*. This function operates like the *renderingCodeToReactElement* function described in the previous section, performing a traversal of the RenderingCode ASTs. For each type of RenderingCode, the function generates the appropriate JavaScript code:

- `HtmlElement` is translated into its corresponding HTML element, attributes, and event handlers.
- `HtmlList` generates Javascript code to dynamically render the list elements, the attributes of the list, and its event handlers.
- `HtmlObject` is translated to its corresponding HTML representation based on the selected *ObjType*, along with its attributes and event handlers.
- `Hole` elements are represented as comments in the generated HTML, maintaining the structure for potential future updates.

A key feature of this function is its ability to automatically populate the generated HTML elements with the corresponding reference to the JSON data, ensuring that the textual representation accurately reflects the created application's structure and content. The references to the JSON data are created dynamically by creating a "path" of the field names during the traversal, which then accurately reflects the structure of the *Model*.

As the user can create custom *messages*, we generate the *Msg* object, which holds the created messages. The *update* function is generated as a JavaScript function and takes three arguments, the *Msg*, the event that triggered the dispatching of this message, and the current application's state. The function consists of a JavaScript *switch statement*, where each case corresponds to a specific message. The body of the cases is specified by the user and inserted without change.

The *startApplications* function is generated the same way for all applications implemented using the *InterfaceSmith* system. It sets the model's value to the input data, provides re-rendering functionality, and creates a dispatch function, which calls the *update* function when a specific UI event is triggered. We see the function along the other parts of the application in Program 15. The result is then inserted into a script HTML element inside an HTML project template.

Program 15 An example Elm-style application generated by the *InterfaceSmith* system.

```
const Msg = {
  MsgExample: "MsgExample",
};

const model = {
  value : 0
};

const update = (msg, event, model) => {
  switch (msg) {
    case Msg.MsgExample:
      return { ...model };
    default:
      return model;
  }
};

const view = (model, dispatch) => `
<div  >
<label  >${model.value}</label>
</div>`;

function startApplication(initialModel, updateFunction, viewFunction
    ) {
  let currentModel = initialModel;
  const render = () => {
    const root = document.getElementById("app");
    root.innerHTML = viewFunction(currentModel, dispatch);
  };
  window.dispatch = (msg, event) => {
    currentModel = updateFunction(msg, event, currentModel);
    render();
  };
  render();
}

startApplication(model, update, view);
```

4.4 Building and Deployment

Now that we have described the implementation of the *InterfaceSmith* system, we must describe how the application is built and deployed. As we mentioned in Section 4.1, we use the *Build* project provided by the *SAFE stack* [34] template. The original SAFE stack *Build* project supports building *full-stack* applications written in F#, and we modified it to remove the support for building the *Shared* and *Server* projects, as our programming system only consists of a client-side application.

4.4.1 Installation and Usage

To install the InterfaceSmith prototype system we provide two main options. The first option is to use the provided pre-built Docker container. The second option is to build the container using the provided *Dockerfile*. After deployment, the system provides two endpoints:

- The InterfaceSmith editor available at:

```
localhost:8080
```

- User and developer documentation available at:

```
localhost:8082
```

Using Pre-built Docker Image

The recommended way to run the InterfaceSmith system is by using the pre-built *Docker* image, as it avoids the lengthy process of downloading dependencies and compiling the application. The only prerequisite is to have Docker installed and running.

To start the InterfaceSmith using the pre-built image:

```
docker load < interfacesmith.tar
docker run -p 8080:8080 -p 8082:8082 interfacesmith
```

Building from Source

Alternatively, the system can be built from source using the following commands:

```
cd InterfaceSmith
docker build -t interfacesmith .
docker run -p 8080:8080 -p 8082:8082 interfacesmith
```

4.4.2 Development Environment

For development purposes, we use the *Nix package manager* [37] to provide a reproducible development environment. The only prerequisite for development is to have the Nix package manager installed and the experimental *Nix flakes* enabled.

To set up the development environment enter the root of the source repository and enter the following set of commands:

```
nix develop
dotnet tool restore
dotnet paket restore
```

The Nix development shell provides all required dependencies:

- .NET SDK 8
- Node.js 20
- Python with mkdocs
- Other development dependencies

4.4.3 Build Tools and Libraries

The SAFE stack *Build* project employs different specialized tools to make building F# application easier. The two main tools used by the *Build* project are the following:

- **FAKE (F# Make):** A build automation tool for projects written in F#, used to define build targets and their individual build steps.
- **Vite:** A web development tool for building JavaScript client-side applications, which provides a development server environment with Hot Module Replacement capability.

The pre-requisites are needed to build the application:

- **.NET Core SDK 8**
- **Node 20**

4.4.4 Build targets

The *Build* project provides multiple different build targets. Each provides a different build result and comprises a different sequence of build steps.

The *Build* project defines the following targets:

- **Run:** Configuration for running a local development server with HMR capabilities. Uses the Fable compiler to compile the source code into JavaScript and uses Vite to run the development server.
- **Bundle:** Target used for bundling of the application for deployment. Uses the Fable compiler to compile the source code into JavaScript and uses Vite to bundle and optimize the application assets, including the JavaScript program, CSS, and static files.
- **RunTests:** Configuration for running the unit tests which are then available at:

  ```
  localhost:8081
  ```

- **Format:** Uses the *Fantomas* F# code formatter, to format the source code.

- **BuildDocs:** Builds the documentation.

In order to build a specific target, we use the following command:

```
dotnet run [Target]
```

It consists of the *dotnet run* command, followed by the name of the target we wish to build. It is important to mention, that the target names are case-sensitive.

4.5 Testing

To ensure that the implemented operations perform as expected, we decided to create a separate project named *Tests*. We implemented *Unit tests* for various operations from the Core Logic module, such as the *replace* function from the *CoreLogic.Operations.RenderingCode* module.

We use the testing library named *Fable.Mocha* [38], which provides useful tools to implement the testing functionality. It provides a browser-based testing environment with a graphical user interface displaying the individual test cases and their results. It also allows composing related *test cases* into *test lists*, improving the readability and maintainability of the tests.

The structure of the individual test cases follows the *Arange Act Assert* (AAA) unit test pattern. It involves preparing the required data, then performing some operation using this data, and then asserting that the operation has the expected result. We can see an example test case in Program 16. We can see the name of the test case and its implementation which follows the *AAA* pattern.

Program 16 An example test case testing the functionality of the *replace* function.

```
testCase "Replace with invalid path"
<| fun _ ->
    let element = RenderingCode.HtmlElement(Tags.div, [], Empty, [])
    let replacement = RenderingCode.HtmlElement(Tags.span, [], Empty
        , [])
    let result = replace [ 0 ] replacement element
    Expect.isError result "Should return error for invalid path"
```

4.6 Summary

In this chapter, we described the practical implementation of our Data-driven UI programming system, translating the theoretical concepts and core logic into a functional prototype. The key points covered in this chapter are:

- In Section 4.1, we outlined the technologies used in our implementation, including F#, Fable, React, Elmish, and others. We also discussed alternative technologies considered and our rationale for our choices.
- In Section 4.2, we presented the system architecture, detailing the modular structure of our implementation. We described the Core Logic and Editor modules, explaining how they interact and their respective responsibilities.
- In Section 4.3, we explored the main features of our system, focusing on:
  - The user interface, which combines an application-level Elm architecture with component-level Elmish architecture for more complex elements.
  - The custom rendering system, which implements the low-code and data-driven approaches through the `renderingCodeToReactElement` function.
  - The code generation capability, which translates our internal representations into HTML and JavaScript.
- In Section 4.4, we described the build and deployment process, detailing the tools used and the available build targets.
- In Section 4.5, we outlined our testing approach, explaining how we use unit tests to verify the correctness of our core operations.

5 Benchmarks

In the previous chapter, we explored the implementation of the *InterfaceSmith* programming system, described the UI and features, and described the code generation capabilities. This chapter evaluates our *InterfaceSmith* prototype system on three tasks: a simple *TO-DO list* application and the *Counter* and *Temperature Converter* from the *7GUIs benchmark* [13]. Our primary goal is to assess whether we can implement these tasks according to their specifications by using only our prototype system and to determine if our approach successfully reduces the amount of code that needs to be written, as per the definition of low-code programming systems by Pinho et al. [3].

5.1 Methodology

We will evaluate our prototype system based on the following criteria:

- **Successful implementation of all specified functionality:** We must implement the exact functionality described by each task. If we cannot implement a specific task as specified, we can reason about our system's limitations and potentially identify problems with our implementation or design.
- **Number of lines of code written:** If a referential solution exists, we will compare the number of lines of code written using our system versus the provided referential solution. We will only consider *physical lines of code* as defined by Park [39]. Because our system requires concrete data before we can begin building the desired application, we will also include the number of lines of code needed to be written for the data preparation. We will also ignore all lines of code related to styling the elements and focus only on implementing the specified functionality and UI structure.

For each task, we will first describe the requirements, such as the intended functionality. After that, we will outline the implementation process using the *InterfaceSmith* system. We will then present the resulting application or describe what functionality cannot be implemented. Lastly, we will analyze the number of lines of code written.

5.2 TO-DO list application

The first task is creating a simple TO-DO list application inspired by the *TodoMVC* [12] benchmark. As our prototype system focuses mainly on exploring the data-driven UI creation and provides limited options for modifying the application's functionality, we will implement a subset of the functionality specified by the *TodoMVC app specification* [40]. As we will implement a subset of the specified functionality, we will not compare the resulting number of LOC to any referential solution.

5.2.1 Task requirements

The TO-DO application's UI will consist of multiple elements. The *InputField* is a text input element describing a new task. *AddTodo* is a button that creates a new task. The *Todo* displays a checkbox and the description of a particular task. The

checkbox represents if the Todo is completed. *Todos* displays the created Todo elements. *CompletedNum* displays the number of completed tasks. Lastly, the *AllDoneButton* is used to mark all Todo elements as completed.

The functionality we will try to implement is the following:

1. The user can input text into the InputField element.
2. When the user clicks the AddTodo and the InputField is not empty, create a new Todo element and display it in the Todos.
3. The CompletedNum reflects how many Todo elements have their checkbox checked.
4. After clicking the AllDone button, all Todo elements become completed.

5.2.2 Creation process

The steps of the creation process are the following:

1. **Prepare JSON data:** The creation process starts by creating JSON data based on the format specified by the task. We can see the created JSON object in Figure 5.1. We model the data based on the UI elements we wish to create according to our specification.

```
{
  "InputField": "",
  "AddTodo": "Add todo",
  "Todos": [
      {
          "text": "Complete project proposal",
          "completed": false
      }
  ],
  "Others": {
      "CompletedCount": 0,
      "AllDoneButton": "All done"
  }
}
```

Figure 5.1 JSON object created as input for the TO-DO list task

2. **Upload data to the system:** We upload the created JSON data to the system.
3. **Replace all hole elements:** We use the provided context menus to replace the hole elements with the new UI elements mirroring the data's structure.
4. **Change the order of the elements:** Using the top-level KeyOrdering menu we change the order of the elements.

5. **Change the tags of the elements:** We use the provided context menus to change the tags for the elements. We select the *input* tag for the InputField and Todo.Completed elements. We choose the *button* tag for the Other.AllDoneButton and AddTodo elements. After that, we choose to make the Todos ordered.

6. **Add necessary attributes:**

   - *Completed element:* We add the *type* attribute to the *Completed* element, select the value as *Constant* and input *checkbox*. Then we add the checked attribute and select the *Data* InnerValue.
   - *InputField element:* We add the *type* attribute with the InnerValue *Constant* set to *text*. After that, we add the *value* attribute and select the *Data* InnerValue.

7. **Create custom Messages:** To implement custom functionality, we use the provided canvas menu for creating messages to create 4 messages which modify the state of the application based on UI events. The application automatically creates the corresponding update function cases for each message.

   - **UpdateInput:** Using the canvas menu, we type the following JavaScript code into the editor window to create the UpdateInput message which we can see in Program 17.

Program 17 Update function case for the UpdateInput message.

```
return {
  ...model,
  InputField: event.target.value
};
```

   - **AddTodo:** Using the canvas menu, we type the following JavaScript code into the editor window to create the AddTodo message which we see in Program 18.

Program 18 Update function case for the AddTodo message.

```
if (!model.InputField.trim()) return model;
return {
  ...model,
  InputField: "",
  Todos: [...model.Todos, {
      text: model.InputField.trim(),
      completed: false
  }]
};
```

   - **ToggleTodo:** Using the canvas menu, we type the following JavaScript code into the editor window to create the ToggleTodo message which we see in Program 19.

Program 19 Update function case for the ToggleTodo message.

```
const todoIndex =
    parseInt(event.target.closest('li').dataset.index);
const updatedTodos = model.Todos.map((todo, index) =>
  index === todoIndex
    ? {...todo, completed: !todo.completed}
    : todo
);
return {
  ...model,
  Todos: updatedTodos,
  Others: {
    ...model.Others,
    CompletedCount: updatedTodos.filter(todo => todo.completed).
        length
  }
};
```

- **CompleteAll:** Using the canvas menu, we type the following JavaScript code into the editor window to create the CompleteAll message which we see in Program 20.

Program 20 Update function case for the CompleteAll message.

```
const allCompleted = model.Todos.map(todo =>
  ({...todo, completed: true}));
  return {
    ...model,
    Todos: allCompleted,
    Others: {
      ...model.Others,
      CompletedCount: allCompleted.length
    }
  };
```

8. **Attach custom handlers to elements:** Using the provided EventHandler menus, we add 4 EventHandlers to 4 of the created elements:

   - We add the *onChange* event with the message handler *UpdateInput* to the InputField element.
   - We add the *onClick* event with the message handler *AddTodo* to the AddTodo element.
   - We add the *onChange* event with the message handler *ToggleTodo* to the Todo.completed element.
   - We add the *onClick* event with the message handler *CompleteAll* to the Others.AllDoneButton element.

5.2.3 Results

We **successfully** created the TO-DO list application and implemented its functionality and UI elements. As we already described the individual update cases based on defined messages, we see the remaining parts generated by the *InterfaceSmith* in Program 21, except the *render* and *init* functions common to all applications generated by our system.

5.2.4 Analysis

As this particular task implements only a subset of the functionality defined by *TodoMVC app specification* [40], we only analyze the number of LOCs needed to implement the desired functionality. We now analyze the Number of LOCs needed to implement the Counter task using our programming system:

- In Step 1, we created the input data and wrote 14 LOC.
- In Steps 2-5, we created and modified the UI elements using only mouse-based operations, resulting in 0 LOC written.
- In Step 6, we added four attributes in total to two different UI elements using the provided modification menus, and we needed to specify Constant InnerValues for 3 of them, resulting in 3 LOC written.
- In Step 7, we created the custom functionality by writing JavaScript code, resulting in 37 LOC written.
- In Step 8, we added the custom handlers to four UI elements using only mouse-based operations, resulting in 0 LOC written.

We wrote **54** lines of physical code in total, and the rest of the operations were performed through a mouse-based interface.

Program 21 The TODO list implementation generated by the *InterfaceSmith* based on the interactions with the system(update, render and init functions not included).

```
const Msg = {
  UpdateInput: "UpdateInput",
  AddTodo: "AddTodo",
  ToggleTodo: "ToggleTodo",
  CompleteAll: "CompleteAll",
};

const Model = {
  InputField: "",
  AddTodo: "Add todo",
  Todos: [
    {
      text: "Complete project proposal",
      completed: false,
    },
  ],
  Others: {
    CompletedCount: 0,
    AllDoneButton: "All done",
  },
};

const view = (
  model,
) => `<div  >
  <input value="${model.InputField}" type="text" onBlur="window.
     dispatch(Msg.UpdateInput, event)" />
  <button  onClick="window.dispatch(Msg.AddTodo, event)">${model.
     AddTodo}</button>

<ul  >${model.Todos.map(
  (item, index) => `
  <li data-index="${index}"><div  ><input type="checkbox" ${item.
     completed ? "checked" : ""}
    onChange="window.dispatch(Msg.ToggleTodo, event)" />
  <span  >${item.text}</span></div></li>`,
).join("")}</ul>

<div  ><button  onClick="window.dispatch(Msg.CompleteAll, event)">${
    model.Others.AllDoneButton}</button>
<div  >${model.Others.CompletedCount}</div></div></div>`;
```

5.3 Counter Task (7GUIs)

The Counter task is defined by the *7GUIs benchmark* [13] as a straightforward task meant to introduce the technology used, its features, and required scaffolding. It consists of two UI elements: a button and a current count label. When the button is clicked, the counter's value is incremented by one.

5.3.1 Task requirements

UI elements

The main UI elements of the application are:

1. Count: A label displaying the current value of the *Count* field.
2. incrementButton: A button used to increment the Count.

Functionality

The functionality we will try to implement is the following:

1. The user can click on the provided button, and the Count is incremented by 1. The Count element is updated to reflect the change in value.

5.3.2 Creation process

The steps of the creation process are the following:

1. **Prepare JSON data:** The creation process starts by creating JSON data based on the format specified by the task. We can see the created JSON object in Figure 5.2.
2. **Upload data to the system:** We upload the created JSON data to the system.
3. **Replace holes with new elements:** This step involves clicking on a provided button menu for each Hole element.
4. **Modify the elements using the context menus:** We use the provided context menus to change the tag of each element. We select the label tag for the Count element and the button for the incrementButton element.
5. **Implementation of custom behavior:** Using the canvas menu, we create the Increment message, specify its name, and type the following JavaScript code into the editor window to implement the message:

Program 22 Update function case for the Increment message.

```
return {
  ...model, Count: model.Count + 1
};
```

6. **Add the EventHandler to the button element:** We add the *onClick* event with the message handler *Increment* to the incrementButton element.

```
{
  "Count": 0,
  "incrementButton": "Increment counter"
 }
```

Figure 5.2 JSON object created as input for the Counter task (7GUIs)

5.3.3 Results

We **successfully** created the desired application and implemented its functionality and UI elements. We can see the code generated by the *InterfaceSmith* system in Program 23.

5.3.4 Analysis

The referential 7GUIs solution [41] for the counter task consists of 11 LOCs written in TypeScript.

We will now analyze the Number of LOCs needed to implement the Counter task using our programming system

- In Step 1, we created the input data and wrote a total of 4 LOC.
- In Steps 2-4, we created and modified the UI elements using only mouse-based operations, resulting in 0 LOC written.
- In Step 5, we created a custom messahe, for which we wrote its name and implementation, resulting in 4 LOC written.
- In Step 6, we added an EventHandler to the `button` element through mouse-based operations, resulting in 0 LOC written.

We wrote **8** lines of physical code in total, and the rest of the operations were performed through a mouse-based interface.

Program 23 The full Counter task implementation generated by the *InterfaceSmith* system.

```
const Msg = {
  Increment: "Increment",
};

const Model = {
  Count: 0,
  incrementButton: "Increment counter",
};

const update = (msg, event, model) => {
  switch (msg) {
    case Msg.Increment:
      return {
        ...model,
        Count: model.Count + 1,
      };

    default:
      return model;
  }
};

const view = (model, dispatch) => `
<div  >
<label  >${model.Count}</label>
<button  onClick="window.dispatch(Msg.Increment, event)">${model.
    incrementButton}</button>
</div>`;

function startApplication(initialModel, updateFunction, viewFunction
    ) {
  let currentModel = initialModel;
  const render = () => {
    const root = document.getElementById("app");
    root.innerHTML = viewFunction(currentModel, dispatch);
  };
  window.dispatch = (msg, event) => {
    currentModel = updateFunction(msg, event, currentModel);
    render();
  };
  render();
}

startApplication(model, update, view);
```

5.4 Temperature Converter Task (7GUIs)

Temperature Converter task is defined by the *7GUIs benchmark* [13] as a more complicated task, as it requires implementing a *bidirectional data flow* between two input elements, and also requires implementing custom validation of user input.

5.4.1 Task requirements

UI elements

The main UI elements of the application are:

1. Celsius: An input element displaying the current value of the *Celsius* field.
2. Fahrenheit: A input element displaying the current value of the *Fahrenheit* field

Functionality

The functionality we will try to implement is the following:

1. The user can change the value of either the Celsius or Fahrenheit elements, and it automatically converts the value of one to the other and updates the elements.
2. If the user inputs a non-numerical value, do not update the other field's value.

5.4.2 Creation process

The steps of the creation process are the following:

1. **Prepare JSON data:** The creation process starts by creating JSON data based on the format specified by the task. We can see the created JSON object in Figure 5.3.

```
{
  "Celsius": "",
  "CelsiusLabel": "Celsius = ",
  "Fahrenheit": "",
  "FahrenheitLabel": "Fahrenheit"
}
```

Figure 5.3 JSON object created as input for the Temperature Converter Task (7GUIs)

2. **Upload data to the system:** We upload the created JSON data to the system.
3. **Replace holes with new elements:** This step involves clicking on a provided button menu for each Hole element.
4. **Modify the elements using the context menus:** We use the provided context menus to change the tag of each element. We select the label tag for the CelsiusLabel and FahrenheitLabel elements and the input tag for the Celsius and Fahrenheit elements.

5. **Implementation of custom behavior:** The implementation of the custom messages is inspired by the referential solution [42]. We define 2 new messages and modify their names.

    - UpdateCelsius update function implementation:

Program 24 Update function case for the UpdateCelsius message.

```
if (isNaN(parseFloat(event.target.value))) {
  return {
    ...model,
    Celsius: event.target.value
  };
}
let celsius = parseFloat(event.target.value);
let fahrenheit = (celsius * 9 / 5 + 32).toFixed(1);
return {
  ...model,
  Celsius: celsius,
  Fahrenheit: fahrenheit.toString()
};
```

    - UpdateFahrenheit update function implementation:

Program 25 Update function case for the UpdateFahrenheit message.

```
if (isNaN(parseFloat(event.target.value))) {
  return {
    ...model,
    Fahrenheit: event.target.value
  };
}
let fahrenheit2 = parseFloat(event.target.value);
let celsius2 = ((fahrenheit2 - 32) * 5 / 9).toFixed(1);
return {
  ...model,
  Fahrenheit: fahrenheit2,
  Celsius: celsius2.toString()
};
```

6. **Add the EventHandlers to the input elements:** We add the *onChange* event with the message handler *UpdateCelsius* to the Celsius element and the *onChange* event with the message handler *UpdateFahrenheit* to the Fahrenheit element.

5.4.3 Results

We *successfully* created the desired application and implemented its functionality and UI elements. We can see the code generated by the *InterfaceSmith* system in Program 26, not including the already described update function.

Program 26 The Counter task implementation generated by the *InterfaceSmith* system(update function not included).

```
const Msg = {
  UpdateCelsius: "UpdateCelsius",
  UpdateFahrenheit: "UpdateFahrenheit",
};

const Model = {
  Celsius: "",
  CelsiusLabel: "Celsius = ",
  Fahrenheit: "",
  FahrenheitLabel: "Fahrenheit",
};

const view = (
  model,
  dispatch,
) =>
`<div  ><input value="${model.Celsius}" onChange="window.dispatch(
    Msg.UpdateCelsius, event)" />
<label  >${model.CelsiusLabel}</label>
<input value="${model.Fahrenheit}" onChange="window.dispatch(Msg.
    UpdateFahrenheit, event)" />
<label  >${model.FahrenheitLabel}</label></div>`;

function startApplication(initialModel, updateFunction, viewFunction
    ) {
  let currentModel = initialModel;
  const render = () => {
    const root = document.getElementById("app");
    root.innerHTML = viewFunction(currentModel, dispatch);
  };
  window.dispatch = (msg, event) => {
    currentModel = updateFunction(msg, event, currentModel);
    render();
  };
  render();
}

startApplication(model, update, view);
```

5.4.4 Analysis

The referential 7GUIs solution [42] for the Temperature Converter task consists of 66 LOCs written in TypeScript using the *React* [1] library.

We will now analyze the Number of LOCs needed to implement the Counter task using our programming system

- In Step 1, we created the input data and wrote a total of 6 LOC.
- In Steps 2-4, we created and modified the UI elements using only mouse-based operations, resulting in 0 LOC written.
- In Step 5, we created a custom function, for which we wrote its name and implementation, resulting in 28 LOC written.
- In Step 6, we added the EventHandlers to both input elements through mouse-based operations, resulting in 0 LOC written.

We wrote **34** lines of physical code in total, and the rest of the operations were performed through a mouse-based interface.

5.5 Evaluation

In this section, we summarize our findings from the benchmark tasks. Our benchmarking process involved implementing three tasks using the *InterfaceSmith* low-code programming system. Our main considerations for this benchmarking are whether we can use our system to implement the specific tasks and how many lines of code we need to write to achieve the desired functionality.

We successfully implemented all three tasks using our programming system, which were the *Counter task*[13], the *Temperature Converter task*[13], and a simple *TO-DO list* application inspired by the *TodoMVC* [12] project. The implementation of each task consisted of obtaining JSON data of a particular structure, uploading the data to the system, creating the UI using the system's low-code context menus, and implementing custom functionality by creating new *message* handlers, which we attached to specific elements triggered by certain events.

The *Counter* task involved creating a simple JSON input, creating the UI using our system's low-code interface, and then implementing a simple update message, which is passed to the update function when the button is clicked, and the state is updated. The task demonstrated that we can create a simple web application using our system, which supports state management and user interaction reactivity.

The *Temperature Converter* task involved creating a simple JSON input, creating the two input fields for the Celsius and Fahrenheit temperatures, and then implementing the desired update functionality based on the user interactions. As explained by *7GUIs benchmark* [13], this task demonstrated that our system provides the necessary tools to implement *bidirectional data flow* between the two input fields.

The *TO-DO list* was the most complex of the selected tasks. It involved creating a JSON input, creating all necessary UI elements, and then implementing the functionality by defining four messages and their implementation. The task demonstrated that we can create more complex applications using the *InterfaceSmith* programming system.

Other than successfully implementing the functionality, our second main consideration is the number of lines of code needed to implement the tasks using our system. Table 5.1 displays the results of the benchmarking process regarding the amount of code we needed to write to implement the custom functionality and UI. Each row of the table corresponds to one of the three benchmarking tasks. The numerical value in the first column indicates the total number of lines of code needed to fully implement the corresponding task. In the second column, the value shows how many lines of code we needed to write in advance to create the input data. The third column shows the number of lines of code needed to implement the specific custom functionality. The third column shows the number of lines of code for each task's referential solution. The fifth column shows whether we successfully implemented the task according to its specification.

Task	Total	Prep	Custom	Ref.	Success
TO-DO List	54	14	40	N/A	Yes
Counter	8	4	4	11	Yes
Temp. Converter	34	6	28	66	Yes

Table 5.1 Implementation Results Summary

The results for the *TO-DO list* task show that we needed to write 54 lines of code in total, where we wrote 14 LOC to prepare the data and 40 LOC to implement the custom functionality. The results for the *Counter* task, displayed on the second row, show that we needed to write 8 LOC in total, 4 of which we wrote to prepare the data and 4 LOC to implement the functionality. For the *Temperature Converter* task, we needed to write 34 LOC in total, and we needed to write 6 LOC to prepare the data and 28 LOC to implement the custom behavior.

Before we compare the amount of LOC needed for each task to their referential solutions, we must state that the referential solutions are implemented in *TypeScript* using the *React* [1] library, which is different from the pure JavaScript our system generates. However, this comparison is intentional, as we want to compare the amount of work needed to implement the tasks using our system to the typical purely text-based implementation approach. The referential solution for the *Counter* tasks is 11 LOC long compared to 8 LOC needed to implement it using our system. The *Temperature Converter* task's referential solution consists of 66 LOC, whereas using our programming system, we only needed to write 34 LOC to implement the same functionality.

Table 5.2 illustrates the code distribution between data preparation and custom functionality implementation across all tasks. Similarly to Table 5.1, each row shows code distribution for one of the tasks. In the first column, we can see the total LOC needed to implement the tasks. In the second column, we can see the percentage of lines of code we wrote to prepare the data compared to the total number of lines of code, The last column shows the percentage of code needed to implement the custom functionality compared to the total number of lines of code. We see that for the *TO-DO list* and *Temperature Converter* tasks, we needed to write considerably more code to implement the custom functionality than the lines of code needed to create the input data.

Task	**Total LOC**	**Data Prep (%)**	**Custom Logic (%)**
TO-DO List	54	26%	74%
Counter	8	50%	50%
Temp. Converter	34	18%	82%

Table 5.2 Implementation Results with Code Distribution Analysis

6 Discussion

In the previous chapter, we implemented three tasks using the *InterfaceSmith* programming system. This chapter will evaluate our work and describe its benefits, limitations, and potential avenues for future research.

Pinho et al. [3] identify several benefits and pitfalls associated with existing programming systems employing the low-code development approach. Some of the common benefits and limitations of existing low-code programming systems are the following:

- **Common Benefits**: The two common benefits of low-code development systems are *Low requirements for technical skills* and *High speed/short development time* while using the programming systems.
- **Common Pitfalls**: Low-code systems commonly suffer from *Interoperability issues* and *Vendor lock-in.*

To better assess the approach's potential viability, we will distinguish between the limitations of our *InterfaceSmith* programming system and the data-driven low-code UI creation approach.

6.1 Evaluation of the approach

This section evaluates the data-driven low-code approach and discusses its positive and negative aspects, separate from our *InterfaceSmith* programming system.

6.1.1 Benefits of the approach

Our work identifies several benefits of the data-driven low-code approach to creating client-side web applications. Each benefit directly results from our specific design principles described in Chapter 2. The described benefits may serve as a motivation to implement new systems employing our described approach or incorporate our approach into existing systems.

The first main benefit is the *guided UI element creation.* The input data inspires the created UI elements and their resulting structure, meaning the user can easily create all necessary UI elements with minimal effort. This process relieves the cognitive load associated with creating UI elements from scratch, as the system automatically creates the right type of element and place it in the correct place in the AST.

The issue of *Vendor lock-in* usually stems from a system's opaque internal representation and the system not allowing the conversion of this internal representation to a more widely used representation, as stated by Pinho et al. [3]. This means the user cannot easily use a different tool to modify or extend the created software components. Our design solves the issue of Vendor lock-in by allowing users to generate a textual representation of the created application.

Another issue our approach does not suffer from is the *lack of interoperability* with other existing systems. Our system accepts input data in a widely used JSON format. The programming system can then be used alongside existing systems, mainly those

focused on data manipulation and retrieval. An example of a programming system that could achieve a synergy with systems implementing our approach is *GraphQL* [32]. Another aspect of our approach that improves interoperability is the ability to generate a textual representation of the created program using a standard representation, which allows the use of other tools that accept the standard representation.

6.1.2 Limitations of the approach

Even though our approach provides several previously described benefits, we also identify several potential limitations. Some of these limitations are a direct result of our design principles described in Chapter 2, while others are a side-effect of our definition of the core logic described in Chapter 3.

As defined in Chapter 2, the data-driven approach first involves providing concrete data to the system. The *necessity to prepare data in advance* may be seen by some users as a potential negative feature of the system, as creating a simple page mandates creating the structured content first. This upfront cost could be lowered by providing simple *data templates*, which the user could modify according to their needs.

The creation process described in Chapter 3 defines a structured way of creating elements based on concrete data. We also define the concept of *structual referencing*, which enables us to dynamically create *Holes* and efficiently combine the created UI elements with corresponding data. However, we must keep the representations' structures precisely the same to ensure the correct structural mapping. The *rigid element structure* prevents us from modifying the created UI elements independently from the corresponding data or creating new UI elements not referencing the created data. To solve this issue, we could explore other options for referencing the data from the UI elements, such as *Selector-based* referencing. This approach involves creating UI elements independently of the input data's structure and referencing it directly. However, we did not explore this option because we wanted to evaluate the purely data-driven approach.

The *incremental creation process* described in Chapter 3 involves simultaneously traversing the Abstract Syntax Trees representing the created UI elements and the input data. During our traversal, we must visit each node of the ASTs to provide the correct modification menus and previews for every created UI element. This means the time to traverse the created AST grows *linearly with the number of elements* they contain. This approach to updating the preview and modification menus could pose a problem when modifying large ASTs, as we perform the traversal after each update of the UI elements. The delay between changing a UI element and seeing the updated result could become too large. This problem can be solved by employing a more targeted approach to re-rendering, such as re-rendering only the modified element and its children.

6.2 Evaluation of the *InterfaceSmith* system

In Chapter 4, we described the implementation of our *InterfaceSmith* programming system. The motivation for creating this prototype system was to show a minimal but cleanly designed system satisfying design principles defined in Chapter 2. In this

section, we evaluate the *InterfaceSmith* system and discuss its positive and negative aspects in implementing the functionality of the data-driven low-code approach.

6.2.1 Positive aspects of the implementation

During our benchmarking process described in Chapter 5, we identified several positive aspects of the *InterfaceSmith* prototype system.

Based on the structure of the supplied concrete data, the system makes it easier to *quickly create user interface elements*. The system provides a modification menu for every created element, allowing for fast property changes via mouse-based operations. After every change, properties are automatically updated, enabling a quick and engaging development loop.

The system provides a *real-time preview* of created elements, automatically populated with corresponding data. This preview updates instantly with each modification, offering immediate visual feedback. A sandboxed preview of the entire web application allows users to test and refine custom behaviors.

The system automatically *generates a complete textual representation* of the created web application. This includes incorporating user-defined custom functionality and generating all necessary HTML elements and JavaScript functions.

6.2.2 Limitations of the implementation

The prototype implementation has several limitations that should be addressed by future research.

The preview and modification menus of *deeply nested data structures are difficult to display* effectively in the current implementation. Due to this limitation, users may find working with complex JSON inputs challenging.

The current implementation supports a *limited set of UI element modification options*. This system does not support all use cases for web application development, particularly for more complex or specialized applications.

The traversal method may cause the *system's performance to deteriorate with large JSON inputs*. This implementation-specific issue could be mitigated with more optimized rendering and update algorithms.

Our prototype system *does not implement persistent storage functionality*. This means all data is kept in memory and lost upon refreshing the page.

These limitations primarily stem from the implementation's prototype nature and the focus on demonstrating the core concepts of the data-driven low-code approach.

6.3 Future work

There are several areas where further research and development could significantly enhance the capabilities and *User Experience (UX)* of low-code data-driven approach and the functionality of the *InterfaceSmith* programming system:

1. **Empirical user studies:** Conducting empirical user studies to evaluate the low-code data-driven approach could help improve the functionality of modifying custom element behavior.

2. **Solving the issue of rigid element structure:** The rigid element structure stems from one of our core design principles. However, the design of the internal representation could be improved to allow for the creation of custom elements independently of the input data.

3. **Improved handling of complex data structures:** Research into more efficient handling of complex input data structures could improve the user experience.

4. **Modification of behavior through a low-code interface:** In our implementation, defining the custom behavior of elements involves writing code in JavaScript. Future research could focus on defining and implementing a low-code interface for implementing the custom behavior without writing code.

5. **Combining data-driven low-code programming with other approaches:** Combining our approach with other existing approaches, such as *output-directed-programming*[8], could improve the functionality of the overall system.

6. **Performance optimization:** Further research into optimizing the rendering and update algorithms could improve the system's performance with large datasets and complex applications.

Conclusion

This thesis explored the *data-driven* approach to creating UI elements using a *low-code* interface, specifically in creating client-side web applications. At the beginning of this thesis, we defined three main *goals* as follows:

1. Explore the applicability of the low-code programming approach coupled with the data-driven approach to creating UI elements of web applications.
2. Create a working prototype programming system named the *InterfaceSmith* implementing the data-driven approach according to the design principles.
3. Benchmark the prototype programming system on three different tasks and analyze the results based on whether the particular tasks can be implemented and how many lines of code we need to write to implement all specified functionality.

To fulfill our first goal, we described several existing systems employing the low-code approach and also systems that combined it with other programming approaches to enhance the development experience, such as *Sketch-and-sketch*[7]. We provided two different definitions of the data-driven approach in Chapter 2, both of which apply to the approach we explored in this thesis. To provide theoretical background behind the data-driven UI element creation, we defined the concept of *structual referencing*, which enables us to dynamically create placeholder *Hole* elements, and efficiently combine the created UI elements with corresponding data. Then we described the mapping between the input data and our internal representation of the UI elements called a *RenderingCode*, and important operations on the RenderingCode type.

To fulfill our second goal, we created a working prototype programming system called the *InterfaceSmith*. The system was implemented according to the design principles in Chapter 2, provides a low-code interface and data-driven UI creation functionality, and generates a textual representation of the created application. The resulting application's textual representation is generated as a pure JavaScript application using the *Elm architecture* [11].

We benchmarked the application on the three tasks we chose at the beginning of the thesis. The tasks were a simple *TO-DO list* application inspired by the *TodoMVC* [12], and the *Counter* and *Temperature Converter* tasks from the *7GUIs benchmark* [13]. We successfully implemented all three tasks according to their specification using the *InterfaceSmith* programming system. We analyzed the amount of code needed to implement the *Counter* and *Temperature Converter* tasks according to their specifications, compared to their referential solutions. We saw that we needed to write less LOC than the referential solution for the two tasks while providing the same resulting functionality.

The positive aspects of the system we noticed during the benchmarking phase were the high speed with which we could create the UI elements based on the input data, the live preview of the created elements the system provides, and the code generation capability. The system's negative aspects include the inability to create new UI elements for which no corresponding input data field exists, potential sub-optimal performance for large input data, and limited availability of UI element modification options.

We successfully fulfilled the goals of our thesis and believe this work provides a stepping stone for further research in low-code data-driven development.

Bibliography

1. *React* [online]. [visited on 2024-04-03]. Available from: https://react.dev/.
2. *Vue.js* [online]. [visited on 2024-04-03]. Available from: https://vuejs.org/.
3. Pinho, Daniel; Aguiar, Ademar; Amaral, Vasco.
What about the usability in low-code platforms? A systematic literature review.
Journal of Computer Languages. 2023, vol. 74, p. 101185. ISSN 25901184.
Available from DOI: 10.1016/j.cola.2022.101185.
4. Sahay, Apurvanand; Indamutsa, Arsene; Di Ruscio, Davide;
Pierantonio, Alfonso. Supporting the understanding and comparison of low-code development platforms. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*.
Portoroz, Slovenia: IEEE, 2020, pp. 171–178. ISBN 9781728195322.
Available from DOI: 10.1109/SEAA51224.2020.00036.
5. *Mendix* [online]. [visited on 2024-05-16].
Available from: https://www.mendix.com.
6. *Darklang* [online]. [visited on 2024-04-22].
Available from: https://darklang.com.
7. Hempel, Brian; Lubin, Justin; Chugh, Ravi.
Sketch-n-Sketch: Output-Directed Programming for SVG. In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*.
New Orleans LA USA: ACM, 2019, pp. 281–292. ISBN 9781450368162.
Available from DOI: 10.1145/3332165.3347925.
8. Chugh, Ravi; Hempel, Brian; Spradlin, Mitchell; Albers, Jacob.
Programmatic and direct manipulation, together at last. *ACM SIGPLAN Notices*.
2016, vol. 51, no. 6, pp. 341–354. ISSN 0362-1340.
Available from DOI: 10.1145/2980983.2908103.
9. Yang, Fan; Gupta, Nitin; Botev, Chavdar; Churchill, Elizabeth F;
Levchenko, George; Shanmugasundaram, Jayavel.
WYSIWYG development of data driven web applications.
Proceedings of the VLDB Endowment. 2008, vol. 1, no. 1, pp. 163–175.
ISSN 2150-8097. Available from DOI: 10.14778/1453856.1453879.
10. *Elm* [online]. [visited on 2024-04-29]. Available from: https://elm-lang.org.
11. *Elm architecture* [online]. [visited on 2024-04-29].
Available from: https://guide.elm-lang.org/architecture.
12. *TodoMVC* [online]. [N.d.]. [visited on 2024-11-18].
Available from: http://todomvc.com.
13. *7GUIs benchmark* [online]. [visited on 2024-04-11].
Available from: https://eugenkiss.github.io/7guis.
14. *JetBrains Rider* [online]. [visited on 2024-11-23].
Available from: https://www.jetbrains.com/rider/.
15. *Github Copilot* [online]. 2024. [visited on 2024-11-23].
Available from: https://github.com/features/copilot.

16. *Visual Studio Code* [online]. [visited on 2024-11-23]. Available from: https://code.visualstudio.com/.

17. *Hugo*. Available also from: https://gohugo.io/.

18. Leonard, Peter. *Macworld Expo 1987 Boston*. 1987. Available also from: http://32by32.com/macworld-expo-1987-boston/.

19. Goodman, Danny. *The complete HyperCard handbook*. 2nd ed. Toronto ; New York: Bantam Books, 1988. The Macintosh performance library. ISBN 9780553345773.

20. Bickell, Scot. *HyperCard emulator* [online]. 1993-03. [visited on 2024-09-17]. Available from: http://archive.org/details/hypercard_guesswords1.

21. Schreiber, Robin; Krahn, Robert; Ingalls, Daniel H. H.; Hirschfeld, Robert. *Transmorphic: Mapping direct manipulation to source code transformations*. Potsdam: Universitätsverlag, 2017. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. ISBN 9783869563879.

22. *Sketch-n-Sketch demo application* [online]. [N.d.]. [visited on 2025-01-01]. Available from: http://ravichugh.github.io/sketch-n-sketch/releases/uist-2019-acm-archive/.

23. *F#* [online]. [visited on 2024-06-11]. Available from: https://fsharp.org.

24. *Fable* [online]. [visited on 2024-04-29]. Available from: https://fable.io.

25. *Feliz* [online]. [visited on 2024-06-11]. Available from: https://zaid-ajaj.github.io/Feliz.

26. Bray, Tim. *The JavaScript Object Notation (JSON) Data Interchange Format* [RFC 8259]. RFC Editor, 2017. Request for Comments, no. 8259. Available from DOI: 10.17487/RFC8259.

27. *Fable.SimpleJson* [online]. [visited on 2024-06-11]. Available from: https://github.com/Zaid-Ajaj/Fable.SimpleJson.

28. Czaplicki, Evan; Chong, Stephen. Asynchronous functional reactive programming for GUIs. *ACM SIGPLAN Notices*. 2013, vol. 48, no. 6, pp. 411–422. ISSN 0362-1340, ISSN 1558-1160. Available from DOI: 10.1145/2499370.2462161.

29. Elliott, Conal; Hudak, Paul. Functional reactive animation. In: *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: Association for Computing Machinery, 1997, pp. 263–273. ICFP '97. ISBN 9780897919180. Available from DOI: 10.1145/258948.258973.

30. Czaplicki. *A Farewell to FRP*. 2024. Available also from: https://web.archive.org/web/20241220190908/https://elm-lang.org/news/farewell-to-frp.

31. *Fable compatibility* [online]. [visited on 2024-04-29]. Available from: https://fable.io/docs/javascript/compatibility.html.

32. *GraphQL* [online]. [visited on 2024-05-16].
Available from: https://graphql.org.

33. *Elmish* [online]. [visited on 2024-06-11].
Available from: https://elmish.github.io.

34. *SAFE stack* [online]. [visited on 2024-04-03].
Available from: https://safe-stack.github.io/.

35. *Thoth.Json* [online]. [visited on 2024-06-12].
Available from: https://thoth-org.github.io/Thoth.Json/.

36. *Tailwind* [online]. [visited on 2024-06-11].
Available from: https://tailwindcss.com/.

37. *Nix package manager* [online]. [visited on 2025-01-01].
Available from: https://nixos.org/.

38. *Fable.Mocha* [online]. [visited on 2024-06-14].
Available from: https://github.com/Zaid-Ajaj/Fable.Mocha.

39. Park, Robert E.
Software Size Measurement: A Framework for Counting Source Statements:
Fort Belvoir, VA, 1992. Available from DOI: 10.21236/ADA258304.

40. *TodoMVC app specification* [online]. 2024-11. [visited on 2024-12-29].
Available from: https://web.archive.org/web/20241119121445/https://github.com/tastejs/todomvc/blob/master/app-spec.md.

41. *7GUIs Example counter implementation* [online]. [visited on 2024-06-23].
Available from: https://github.com/eugenkiss/7guis-React-TypeScript-MobX/blob/master/src/app/guis/counter.tsx.

42. *7GUIs Example temperature converter implementation* [online].
[visited on 2024-06-23].
Available from: https://github.com/eugenkiss/7guis-React-TypeScript-MobX/blob/master/src/app/guis/tempconv.tsx.

List of Figures

List of Code Examples