GUIDE

Graphical User Interface Development Environment

Andrea Longoni

Id. Number: 13896A

MSc in Computer Science

Advisor: Prof. Walter Cazzola
Co-Advisor: Dr. Matteo Brancaleoni

UNIVERSITÀ DEGLI STUDI DI MILANO
Computer Science Department
ADAPT-Lab

Academic Year 2023–2024

COMPUTER SCIENCE DEPARTMENT • ADAPT-LAB

Contents

1 Introduction

Graphical User Interfaces (GUIs) play a crucial role in modern software applications, serving as the primary medium through which users interact with digital systems. The design and implementation of GUIs require significant effort, often involving repetitive coding tasks and adherence to complex framework-specific rules. As applications grow in complexity, ensuring consistency, maintainability, and adaptability of GUIs across different platforms becomes increasingly challenging.

To address these challenges, the software engineering community has explored various approaches to streamline GUI development. One widely adopted strategy is the use of code generation [18], where high-level specifications are automatically transformed into executable code. Code generation techniques reduce manual effort, minimize human errors, and promote consistency across different implementations. However, traditional code generation approaches often lack flexibility, making it difficult to customize or extend the generated UI components.

Another approach to improving GUI development is the use of **Domain-Specific Languages (DSLs)** [17]. A DSL is a programming language or specification language designed for a particular domain, offering high-level abstractions that simplify the development process. In the context of GUI development, DSLs allow designers and developers to describe UI structures declaratively, focusing on the layout and behavior rather than low-level implementation details. This abstraction enhances productivity and promotes separation of concerns between UI definition and application logic.

In parallel, **Software Product Lines (SPLs)** [3] have emerged as a powerful paradigm for managing variability and reuse in software systems. SPLs enable the systematic production of multiple software variants by leveraging a common set of core assets while allowing configurable customizations. When applied to GUI development, SPL principles facilitate the modularization of UI components, enabling developers to assemble different interface configurations based on specific requirements.

Despite these advancements, existing solutions often struggle to balance flexibility, modularity, and maintainability. Many code generation tools produce monolithic outputs that are difficult to modify, while some DSL-based solutions remain tightly coupled to specific frameworks, limiting their adaptability. Additionally, SPL techniques, while effective in managing variability, are rarely applied to GUI development in a way that seamlessly integrates with DSL-based approaches.

This thesis addresses these challenges by proposing a novel methodology that combines DSLs and SPLs for GUI generation. By leveraging the strengths of both paradigms, the proposed approach aims to provide a flexible, modular, and reusable solution for

defining and generating UI code across multiple programming languages. The following sections outline the limitations of current technologies, the objectives of this research, and the contributions of the proposed framework.

Limitations of Existing Technologies. Despite the numerous advancements in GUI development, existing technologies present several limitations that hinder flexibility, maintainability, and scalability. Traditional GUI frameworks and code generation tools often impose rigid structures that make adaptation to different contexts challenging. These limitations can be categorized into three main areas: lack of modularity, limited support for multiple target languages, and inefficiencies in code reuse.

One significant drawback of many GUI development approaches is the **lack of modularity**. In several existing frameworks, UI components are tightly coupled with their underlying implementation, making it difficult to selectively include or exclude specific features. This monolithic approach results in large, unwieldy codebases where even minor modifications require extensive changes across multiple files. Additionally, many frameworks do not provide a structured way to manage feature variability, limiting their applicability in scenarios requiring high levels of customization.

Another major limitation concerns the **restricted support for multiple target languages**. Many GUI frameworks are designed with a specific programming language in mind, forcing developers to rewrite UI logic when porting applications to different platforms. Even in cases where cross-platform solutions exist, such as Flutter[1] or Qt[2], these frameworks typically impose their own ecosystem, requiring developers to conform to their APIs and rendering engines. This constraint reduces the flexibility of software teams that may need to integrate with existing codebases in different languages.

Furthermore, **code reuse and maintainability** remain persistent challenges. While modern software engineering practices encourage component reuse, many GUI solutions lack an effective way to share UI components across different projects or configurations. When developers need to support multiple variants of an application, they often resort to duplicating and modifying code manually, leading to increased maintenance effort and higher risks of inconsistencies. The absence of systematic mechanisms for handling variations in UI structures further exacerbates this issue.

The shortcomings of these technologies highlight the need for a new approach that combines the strengths of multiple paradigms. By integrating **Domain-Specific Languages (DSLs)** and **Software Product Lines (SPLs)**, it is possible to address these limitations, offering a more modular, adaptable, and reusable way to define and generate GUIs. The next section introduces the key objectives of this research, outlining how the proposed approach aims to overcome these challenges.

Contributions. This research introduces **GUIDE (Graphical User Interface Development Environment)**, a system designed to *guide* developers in the creation of graphical

[1] https://flutter.dev/
[2] https://www.qt.io/

user interfaces (GUIs) by automating code generation and providing high-level abstractions. By combining **Domain-Specific Languages (DSLs)** and **Software Product Lines (SPLs)**, GUIDE streamlines the UI development process across multiple programming languages.

The key contributions of this work are as follows:

- **Definition of a DSL for UI specification**: GUIDE provides a domain-specific language designed to simplify the definition of user interfaces. The DSL enables developers to describe UIs in a high-level, declarative manner, abstracting away the complexities of underlying implementation details.
- **Automated multi-platform code generation**: GUIDE supports the generation of UI code for multiple target languages, eliminating the need for developers to manually implement the same interface in different programming environments.
- **A modular and configurable architecture**: GUIDE applies SPL principles to organize UI components, layouts, and features as independent modules. This modularity allows developers to generate customized versions of the system, including only the required components and reducing unnecessary dependencies.
- **Enhanced maintainability and reusability**: By structuring UI definitions within a DSL and leveraging SPL-based modularization, GUIDE facilitates the reuse of UI components across different projects while maintaining consistency and reducing code duplication.
- **Improved distribution efficiency**: GUIDE incorporates optimizations to minimize the impact of large dependencies, allowing developers to generate lightweight versions of the system tailored to their specific needs.

These contributions aim to improve efficiency in GUI development, reduce manual coding effort, and provide a scalable approach to UI generation. The following section outlines the structure of this thesis, detailing how each chapter addresses these research contributions.

Outline. This thesis is structured to provide a comprehensive understanding of GUIDE, progressing from its conceptual foundations to its implementation and evaluation. The discussion begins with an overview of the fundamental concepts and technologies that underpin the project. Chapter 2 introduces Domain-Specific Languages (DSLs), Software Product Lines (SPLs), and code generation techniques, establishing the theoretical framework that justifies the approach taken in GUIDE. These concepts form the foundation for the following chapters, where their practical application is explored in depth.

Building on this background, Chapter 3 presents the challenges faced in UI development and code generation, particularly concerning modularity, language support, and maintainability. The chapter highlights the limitations of existing solutions and defines the research problem that GUIDE aims to address. This problem statement sets the stage for Chapter 4, which details the proposed architecture of GUIDE. Here, the modular structure of the system is introduced, describing how its DSL, library, and SPL

components interact to enable flexible and scalable UI generation.

Following the architectural design, Chapter 5 delves into the technical details of GUIDE's implementation. This chapter explores the core elements of the DSL and the supporting library, including key components, syntax definitions, and the mechanisms that facilitate code generation. To provide a concrete understanding, relevant code listings and diagrams illustrate how GUIDE transforms high-level UI descriptions into executable code.

Once the implementation has been described, Chapter 6 evaluates the effectiveness of GUIDE through a series of experiments. This chapter measures GUIDE's impact on code reduction, development effort, and storage efficiency, comparing its benefits to traditional UI development approaches. The results of this analysis serve to validate the effectiveness of GUIDE in real-world scenarios.

In Chapter 7, the thesis situates GUIDE within the broader context of research on DSLs, SPLs, and automated GUI generation. The discussion reviews existing academic literature and compares GUIDE to similar approaches, highlighting both its innovations and its alignment with prior work in the field.

Finally, Chapter 8 summarizes the main contributions of this research, discussing the strengths and limitations of GUIDE. The chapter also explores potential directions for future work, such as expanding GUIDE's DSL capabilities, enhancing its architecture, and improving usability through new development tools. This final discussion reflects on the broader impact of GUIDE and outlines areas for continued refinement and exploration.

Overall, this thesis follows a structured approach, guiding the reader from theoretical foundations to practical implementation and empirical validation. By presenting the motivations, contributions, and evaluation of GUIDE in a clear and progressive manner, it provides a complete perspective on the system's role in addressing the challenges of UI development.

2 Background

2.1 Domain-Specific Languages (DSLs)

Languages are the foundation of software development, serving as the medium through which developers express instructions, designs, and ideas to machines. Traditionally, General-Purpose Languages (GPLs) such as Java[1], Python[2], and C++[3] have been widely used to develop a vast range of applications due to their flexibility and broad applicability. However, the increasing complexity of software systems has driven the need for languages tailored to specific tasks or domains. Domain-Specific Languages (DSLs) emerge as a response to this need, providing a means to describe problems and solutions within a constrained area of expertise.

DSLs are not intended to solve a wide range of computational problems. Instead, they focus on providing a highly specialized, expressive, and efficient syntax for specific tasks. This specialization makes DSLs particularly valuable in domains where precision, readability, and abstraction are critical, such as configuration management, query processing, and user interface design.

The evolution of DSLs reflects the growing importance of bridging the gap between domain experts and software developers. By aligning closely with domain concepts, DSLs allow non-programmers to interact with software systems effectively while enabling developers to focus on broader architectural concerns. This synergy has made DSLs a powerful tool in modern software engineering.

2.1.1 A comparison between DSLs and GPLs

Domain-Specific Languages (DSLs) are specialized languages designed to address specific tasks within a defined domain. Unlike General-Purpose Languages (GPLs), which are versatile and can handle a broad range of programming challenges, DSLs focus on providing precise and efficient solutions for domain-specific problems. This distinction shapes the way these languages are designed, used, and maintained.

DSLs excel in scenarios where domain experts need to interact with software systems. Their syntax and semantics are closely aligned with the domain, enabling non-programmers to express complex ideas without extensive knowledge of programming. For instance, SQL [14] allows database administrators to manipulate and query

[1] https://www.java.com/
[2] https://www.python.org/
[3] https://www.cplusplus.com/

data without delving into the complexities of database engine internals. Conversely, GPLs such as Python or Java require broader programming expertise, as they are designed to solve problems across diverse domains.

However, the specialization of DSLs comes at a cost. While GPLs are supported by extensive ecosystems of libraries, tools, and communities, DSLs often have limited tooling and are less portable. This trade-off between focus and flexibility defines the roles of DSLs and GPLs in software development.

2.1.2 Advantages and Limitations of DSLs

The primary advantage of DSLs lies in their ability to abstract away unnecessary complexity. By narrowing their focus to a specific domain, they provide expressive and concise syntax that reduces both development time and error rates. Their alignment with domain concepts also improves readability and maintainability, especially for domain experts who are not professional programmers.

On the other hand, this focus limits their scope of applicability. A DSL is often unsuitable for tasks outside its intended domain, necessitating integration with GPLs for broader functionality. Furthermore, developing and maintaining a DSL can be challenging, particularly when it involves designing custom tooling, ensuring compatibility with other systems, or adapting to evolving domain requirements.

2.1.3 Internal and External DSLs

DSLs can be classified into two main categories: internal and external.

Internal DSLs are embedded within a host GPL, leveraging its syntax, semantics, and runtime environment. They often appear as libraries or frameworks, offering a DSL-like syntax while benefiting from the ecosystem and tooling of the host language. For example, testing frameworks or query libraries embedded in programming languages exemplify this approach.

External DSLs, in contrast, are standalone languages with their own syntax, semantics, and execution environments. They require dedicated parsers or compilers to translate their code into executable artifacts. While external DSLs offer greater flexibility in designing language constructs, they demand significant effort to develop and maintain, including the creation of custom editors, debuggers, and other tools.

A practical example of this distinction can be seen in database querying. SQL itself is an external DSL, as shown in Listing 2.1. It has its own syntax and execution model, requiring a dedicated parser and interpreter within a database engine.

```sql
SELECT name, age FROM users WHERE age > 18 ORDER BY name;
```

Listing 2.1. *Example of an external DSL: SQL.*

On the other hand, many programming languages provide internal DSLs for database interaction, embedding query-like functionality within their syntax. For example, in

Python using SQLAlchemy[4] (Listing 2.2), the query is expressed using an object-oriented API within Python, leveraging the language's syntax while abstracting SQL operations.

```
session.query(User).filter(User.age > 18).order_by(User.name).all()
```

Listing 2.2. *Example of an internal DSL: SQLAlchemy in Python.*

The choice between internal and external DSLs depends on the balance between the desired level of expressiveness and the cost of implementation. Internal DSLs are often simpler to create but constrained by the syntax and semantics of the host language, whereas external DSLs allow for complete customization at the expense of higher development costs.

2.1.4 Syntax and Semantics in DSLs

The design of a DSL involves three foundational aspects: abstract syntax, concrete syntax, and semantics. The abstract syntax defines the logical structure of the language, focusing on the relationships and hierarchies between its constructs. It is typically represented using structures such as abstract syntax trees, which serve as the backbone of the language's grammar.

The concrete syntax, on the other hand, specifies how the language is represented textually or visually. This includes the keywords, symbols, and formatting rules that users interact with directly. While abstract syntax defines the underlying structure, the concrete syntax shapes the user experience, influencing the ease of learning and using the language.

Finally, semantics provide meaning to the language constructs. They define how each element of the DSL behaves, either through operational rules that describe execution or through mathematical mappings that establish formal properties. Semantics ensure that the language's syntax aligns with the intended domain, enabling precise and predictable behavior.

2.2 Code Generation

Code generation is the process of automatically producing code from higher-level specifications, reducing repetitive tasks, minimizing errors, and improving development efficiency. It enables the transformation of abstract models into executable programs or intermediary artifacts, streamlining software development.

There are multiple approaches to code generation, each catering to different use cases. Two widely used techniques are source code generation and template-based generation, which offer varying degrees of automation and flexibility.

[4] https://www.sqlalchemy.org/

2.2.1 Source Code Generation

Source code generation involves producing full or partial implementations in general-purpose programming languages such as Java, Python, or C++. This technique is particularly useful for eliminating boilerplate code, automating repetitive patterns, and ensuring consistency across large projects.

Many modern frameworks leverage source code generation to create foundational components like data models, controllers, or configuration files. This accelerates development by providing a structured starting point while allowing customization when needed.

For example, web frameworks often include tools to generate standardized components such as database models and API endpoints, reducing manual effort. The generated code serves as a base that developers can refine and extend according to project-specific requirements.

By automating the creation of frequently used structures, source code generation enhances productivity, maintains uniformity, and reduces development overhead, particularly in large-scale or structured environments.

2.2.2 Template-Based Generation

Template-based generation is a popular technique that involves using predefined templates to generate code automatically. Templates are essentially blueprints that define the structure and behavior of the generated code, but include placeholders that are filled in with specific values or logic during the generation process.

This approach is particularly useful in scenarios where the structure of the generated code follows predictable patterns. Templates can be designed to create complex systems or simple components such as configuration files, database schemas, or API endpoints. By reusing these templates, developers can automate the creation of large portions of code that would otherwise be tedious and repetitive to write by hand.

Template-based generation works by defining a set of rules and patterns that describe the structure of the code. These patterns are then applied to specific data inputs or configurations to generate the desired code. This method can be applied across a wide range of use cases, and templates can be modified or extended to suit new requirements.

2.3 Software Product Line Engineering

A Software Product Line (SPL) is a systematic approach to software development that focuses on creating a family of related software products. Unlike traditional methods, which often treat each software product as a standalone entity, SPL aims to leverage commonalities among products while managing variability to meet specific requirements. This approach is particularly valuable in contexts where software systems need to be customized for different clients, markets, or operational environments.

The concept of SPL emerged as a solution to the challenges posed by the growing complexity and diversity of software systems. By emphasizing reuse and modularity, SPL enables organizations to develop multiple product variants efficiently, reducing time-to-market and overall development costs. At its core, an SPL consists of a core asset base, which includes reusable components, architectures, and processes, and a mechanism to configure these assets into specific product variants.

Key to the success of SPL is the ability to manage variability, which refers to the differences among products within the product line. Variability is captured and managed through techniques such as feature modeling, which provides a structured representation of the options and choices available in the product family. Feature models allow stakeholders to define and configure product variants systematically, ensuring that the resulting systems meet their requirements while maintaining consistency across the product line.

SPL has gained significant traction in various domains, including automotive software, embedded systems, and enterprise applications, where the need for tailored solutions is critical. By adopting SPL, organizations can achieve significant benefits such as improved productivity, enhanced quality, and better scalability. However, implementing SPL also presents challenges, particularly in managing the trade-offs between flexibility and complexity.

2.3.1 SPL Architecture and Artifacts

The architecture of a Software Product Line (SPL) is a carefully designed structure that enables the systematic reuse of shared assets while accommodating the variability needed to produce diverse products. This architecture is central to the efficient operation of SPLs, as it ensures that shared resources are effectively managed and variability is seamlessly integrated into the product generation process.

At the foundation of an SPL lies the core asset base [23], a repository that houses all reusable elements necessary for building the product line. These assets can include software components, libraries, templates, and documentation, all designed to be shared across multiple products. The core asset base is meticulously curated to ensure consistency and maintainability, making it a cornerstone of the SPL approach. By centralizing reusable assets, organizations reduce duplication of effort, streamline development processes, and maintain high standards of quality across their product offerings.

A crucial aspect of SPL architecture is modularization, which facilitates the management of variability. By dividing the system into modular components, developers can address specific requirements without impacting the overall architecture. Each module represents a self-contained unit of functionality that can be independently customized or replaced. This modular approach not only simplifies the integration of variability but also enhances scalability, enabling the SPL to grow and evolve as new features or market demands arise. Techniques such as feature-based composition and aspect-oriented programming are commonly employed to implement modular architectures, ensuring that the variability defined in the SPL can be effectively managed and deployed.

Another vital component of SPLs is the generation of product-specific artifacts, which translate the variability defined in the feature model into tangible outputs. These artifacts may include source code, configuration files, and executable binaries, all tailored to the unique configuration of a particular product. Automation plays a significant role in this process, with tools that interpret the feature model, resolve dependencies, and assemble the required components from the core asset base. By automating artifact generation, organizations minimize the risk of errors and ensure that the final products align with the defined requirements and constraints.

2.4 Neverlang: A Language Workbench for DSLs

Neverlang [11, 13, 29, 30] is a JVM-based language workbench [16] designed to facilitate modular language development by enabling the composition and reuse of language features. Unlike traditional monolithic approaches to compiler and interpreter design, Neverlang allows language constructs to be defined as independent, composable modules that can be combined dynamically to create a fully functional programming language or domain-specific language (DSL).

The framework follows a syntax-directed translation approach [1], in which a compiler or interpreter constructs an abstract syntax tree (AST) from an input program and processes it through multiple compilation phases. Each phase visits the AST and applies transformations or evaluations to its nodes, ultimately producing executable code or performing an interpretation of the program.

2.4.1 Modular Language Implementation

The core principle of Neverlang is modularity. Each language feature is encapsulated in a self-contained module, which consists of:

- A **syntax definition**, typically expressed using a BNF-like notation.
- One or more **roles**, each defining a specific semantic action or compilation phase.

Neverlang introduces the concept of slices, which act as grouping mechanisms for language features. A slice can aggregate syntax rules and roles from multiple modules, effectively enabling different parts of a language to be developed separately and later combined into a complete language specification.

2.4.2 Extensibility and Composition

A key strength of Neverlang is its ability to evolve languages incrementally by composition [12]. New features can be added without modifying the existing implementation, simply by introducing new modules and slices. Additionally, Neverlang's design allows language implementations to be distributed as precompiled components, reducing the need for recompilation when extending an existing language.

The framework also supports language variability, making it possible to define different configurations of a language tailored to specific use cases. This aligns well

with the principles of Software Product Lines (SPLs), as developers can selectively include only the necessary features when generating a language variant.

2.4.3 Application to Domain-Specific Languages

Neverlang is particularly well-suited for the development of DSLs, as it provides mechanisms to define domain-specific constructs in a modular and maintainable way. Its ability to integrate with Java and other JVM-based languages makes it a practical choice for DSL implementations that require seamless interoperability with existing software ecosystems.

In summary, Neverlang provides a highly flexible and modular approach to language development, enabling composable language features, incremental extensibility, and efficient reuse of language components. Its design aligns closely with modern software engineering practices, making it a powerful tool for building both general-purpose and domain-specific languages.

2.5 Graphical User Interfaces (GUIs)

Graphical User Interfaces (GUIs) are a fundamental component of modern software systems, providing users with intuitive ways to interact with applications and access their functionality. GUIs present information visually through graphical elements such as windows, buttons, menus, and icons, enabling users to perform tasks efficiently and effectively.

The design of GUIs is a multidisciplinary field that combines principles of human-computer interaction, graphic design, and software engineering. A well-designed GUI enhances user experience by providing clear navigation, consistent feedback, and intuitive controls. GUIs are essential in a wide range of applications, from desktop software and web applications to mobile devices and embedded systems.

The development of GUIs involves several key considerations, including layout design, interaction patterns, accessibility, and responsiveness. Designers must balance aesthetic appeal with usability, ensuring that the interface is visually engaging while remaining functional and accessible to diverse user groups. The choice of graphical elements, color schemes, typography, and animations all contribute to the overall user experience.

GUIs can be created using a variety of technologies and frameworks, each offering different levels of flexibility, performance, and platform compatibility. Common GUI development tools include libraries such as Qt, JavaFX[5], and GTK[6], as well as web technologies like HTML [8], CSS [26], and JavaScript[7].

For modern web-based UI development, frontend frameworks such as Bootstrap[8] and

[5] https://openjfx.io/
[6] https://www.gtk.org/
[7] https://js.org/
[8] https://getbootstrap.com/

Tailwind CSS[9] provide predefined styles and responsive design capabilities, reducing the need for extensive manual styling. Additionally, server-driven UI frameworks like Phoenix LiveView[10], built on Elixir[11], enable real-time web applications by maintaining a persistent connection between the client and server, minimizing JavaScript usage while improving performance and maintainability.

The choice of technology depends on factors such as the target platform, performance requirements, and developer expertise.

2.5.1 Key Concepts of GUIs

At its core, a GUI is a visual interface that allows users to interact with software through graphical elements such as buttons, menus, icons, and windows, rather than relying solely on text-based commands. The evolution of GUIs has been pivotal in making computing accessible to non-technical users, transforming software systems into tools that can be efficiently used in various domains, from personal computing to industrial applications.

GUIs are often contrasted with Command-Line Interfaces (CLIs), which require users to input textual commands to interact with a system. While CLIs offer efficiency and precision for advanced users, GUIs prioritize user-friendliness and accessibility. They rely on direct manipulation principles, where users interact with visual representations of objects to perform actions, such as dragging a file to delete it or clicking a button to submit a form.

2.5.2 Architectural Components of GUIs

The design and development of GUIs follow well-defined architectural principles that separate the user-facing elements from the underlying application logic. This separation promotes modularity, scalability, and ease of maintenance. The primary architectural components of GUIs include:

- **Presentation Layer**: This layer defines the visual elements that users interact with, such as buttons, sliders, text fields, and graphical layouts. It focuses on the aesthetics and usability of the interface, ensuring that it is intuitive and visually appealing.
- **Event Handling Mechanism**: GUIs are event-driven systems where user interactions, such as clicks or key presses, trigger events. The event handling mechanism processes these inputs and ensures that the appropriate application logic is executed in response.
- **Application Logic**: This layer contains the underlying functionality of the software, decoupled from the interface itself. For instance, clicking a "Save" button might trigger a function in the application logic to write data to a file.

[9] https://tailwindcss.com/
[10] https://www.phoenixframework.org/
[11] https://elixir-lang.org/

- **Data Model**: GUIs often interact with a data model that represents the application's state. The synchronization between the GUI and the data model is crucial for providing real-time feedback and ensuring that user actions are accurately reflected.

2.6 Design Patterns

Design patterns [19, 20] are reusable solutions to common problems encountered in software design and development. They provide a structured approach to solving recurring challenges, offering proven solutions that can be adapted to different contexts. Design patterns help developers create software that is modular, flexible, and maintainable, by capturing best practices and design principles in a reusable format.

2.6.1 Adapter Pattern

The Adapter Pattern is a structural design pattern that allows incompatible interfaces to work together. It enables communication between two incompatible interfaces, allowing them to collaborate without modifying their existing code. The Adapter Pattern is particularly useful when integrating legacy systems, third-party libraries, or components with different interfaces.

The key components of the Adapter Pattern are:

- **Target Interface**: The interface that the client code expects to interact with. This is the interface that the client code is designed to work with.
- **Adaptee Interface**: The interface of the existing component that needs to be adapted. This interface is incompatible with the target interface.
- **Adapter**: The class that connects the target interface and the adaptee interface, allowing them to work together. It translates calls from the target interface into calls to the adaptee interface, ensuring that the two can work together seamlessly.

The Adapter Pattern is commonly used in scenarios where components with different interfaces need to collaborate, such as when integrating external services, libraries, or modules into an existing system. By providing a layer of abstraction that translates between interfaces, the Adapter Pattern promotes interoperability and reusability without requiring extensive modifications to existing code.

2.6.2 Composite Pattern

The Composite Pattern is a structural design pattern that allows clients to treat individual objects and compositions of objects uniformly. It enables the creation of hierarchical structures where individual objects and groups of objects can be manipulated in a consistent manner. The Composite Pattern is particularly useful when dealing with tree-like structures or recursive compositions.

The key components of the Composite Pattern are:

- **Component**: The interface or abstract class that defines the common operations for both individual objects and compositions. This component represents the building block of the composite structure.
- **Leaf**: The individual objects that make up the composite structure. These are the basic elements that do not have any children.
- **Composite**: The class that represents the composite structure, composed of individual objects and other composite objects. The composite class implements the operations defined in the component interface and manages the collection of child components.

The Composite Pattern simplifies the manipulation of complex structures by providing a unified interface for interacting with individual objects and compositions. Clients can treat all elements uniformly, regardless of their specific type, enabling recursive operations and traversal of hierarchical structures.

2.6.3 Abstract Factory Pattern

The Abstract Factory Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It ensures that a group of objects created together are compatible and consistent, promoting flexibility and scalability in software design.

The key components of the Abstract Factory Pattern are:

- **Abstract Factory**: Declares a set of methods for creating each type of product in a family, ensuring consistency among them.
- **Concrete Factory**: Implements the abstract factory interface, providing specific implementations for each product defined by the abstract factory.
- **Abstract Product**: Defines the common interface for a family of products, guaranteeing that all created objects share a consistent interface.
- **Concrete Product**: Implements the abstract product interface, representing the actual objects that are created by the concrete factory.

The Abstract Factory Pattern promotes loose coupling between the client code and the concrete classes of the products. This allows new families of products to be introduced without modifying existing code, enhancing maintainability and scalability. It is particularly useful when a system needs to be configured with one of several families of related objects, ensuring that all products within a family are used together consistently.

2.6.4 The Builder Pattern

The **Builder Pattern** is a creational design pattern that provides a flexible and structured way to construct complex objects step by step. It is particularly useful when an object requires multiple optional parameters or configurations, allowing clients to specify only the attributes they need while maintaining a readable and fluent interface.

The key components of the Builder Pattern are:

- **Builder Interface**: Defines the steps required to construct the object, typically providing methods to set various attributes.
- **Concrete Builder**: Implements the builder interface, storing the configuration and assembling the final object.
- **Product**: The complex object being created, which may require multiple optional parameters.
- **Director (Optional)**: An optional component that orchestrates the construction process, ensuring that objects are built following a predefined sequence.

The Builder Pattern improves code maintainability by separating the creation logic from the object itself, reducing constructor complexity and enhancing readability. It is commonly used in scenarios where objects have numerous optional attributes, such as UI components, configuration objects, and data structures with hierarchical relationships.

3
Problem Statement

The development of graphical user interfaces (GUIs) is a fundamental aspect of modern software engineering, bridging the gap between complex systems and end-users through intuitive and visually appealing interaction mechanisms. However, despite their importance, GUI development often poses significant challenges for developers, particularly in scenarios that demand multi-platform compatibility, modular design, and support for various programming languages.

Traditional approaches to GUI design typically involve a significant amount of manual coding tailored to specific frameworks or languages, resulting in high development effort, limited reusability, and increased potential for errors. Moreover, adapting GUIs to different programming environments further complicates the process. These challenges are particularly pronounced in projects requiring flexibility and scalability, where changes in requirements can necessitate substantial rewrites or modifications to the existing codebase.

To address these issues, the field of code generation has emerged as a promising approach. By automating the creation of executable code from high-level specifications, code generation significantly reduces development overhead, improves consistency across projects, and minimizes human error. Domain-Specific Languages (DSLs) are particularly well-suited for this task, as they allow developers to express solutions in terms of domain-specific abstractions, thereby encapsulating complexity and enhancing productivity.

However, existing tools and frameworks for code generation often suffer from critical limitations. They may lack the flexibility to accommodate diverse application domains, provide insufficient modularity for reusing components, or impose constraints on the languages or platforms they support. This gap highlights the need for a solution that combines the expressiveness of DSLs with the configurability of Software Product Lines (SPLs) to generate modular, language-agnostic GUI code tailored to specific project requirements.

3.1 Existing Technologies and Limitations

In the domain of graphical user interface (GUI) development and code generation, numerous technologies and frameworks have emerged to simplify the process and reduce manual coding effort. These tools typically offer solutions for creating and managing GUIs in specific programming environments, streamlining workflows, and

generating code. However, despite their contributions, existing technologies often fall short in addressing the needs of modern, highly modular, and language-agnostic systems.

3.1.1 Slint: A Declarative Toolkit for GUI Development

Slint[1] is a modern toolkit designed to streamline the development of graphical user interfaces (GUIs) through a declarative and cross-platform approach. By providing a dedicated domain-specific language (DSL) and focusing on efficiency and simplicity, Slint addresses many common challenges in GUI creation, such as reducing manual coding effort and ensuring compatibility across diverse platforms. Its design goals are centered on **S**calable, **L**ightweight, **I**ntuitive and **NaT**ive GUI development.

Overview and Architecture

At its core, Slint employs a declarative DSL to define the structure, layout, and behavior of GUIs. This language allows developers to describe interfaces using high-level abstractions, reducing the need for imperative programming. The DSL abstracts away the complexities of underlying GUI frameworks, letting developers focus on design and logic rather than low-level implementation details.

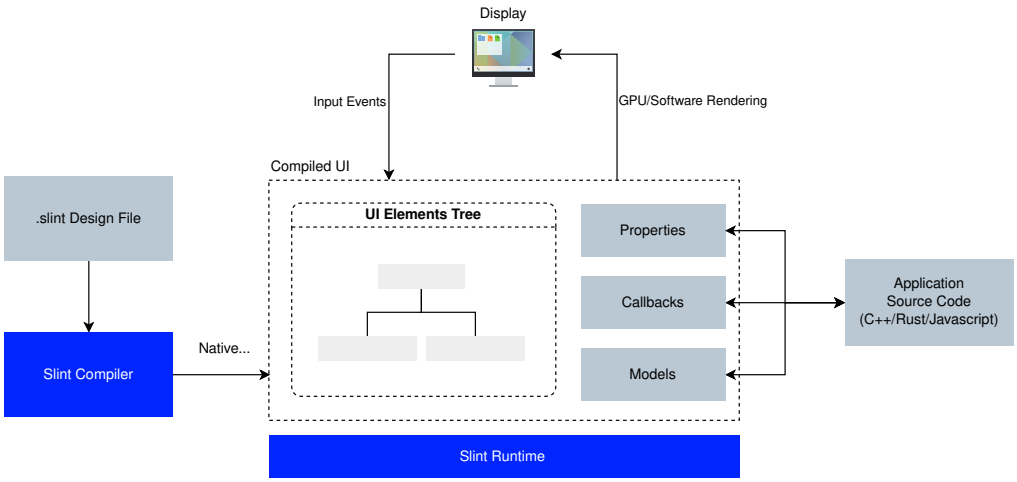

Figure 3.1. *Slint's architecture is built around three primary components: the DSL Compiler, Runtime Engine, and Backend Adapters. The DSL Compiler translates high-level interface definitions into executable code, the Runtime Engine manages dynamic aspects of GUIs, and Backend Adapters enable integration with different platforms .*

Slint's architecture is built around three primary components as shown in Figure 3.1:

[1] `https://slint.dev/`

- **The DSL Compiler**: Translates high-level interface definitions into code that can be executed by supported backends. This ensures that GUIs are consistently rendered across platforms.
- **Runtime Engine**: Manages the dynamic aspects of GUIs, such as animations, user interactions, and state changes. It uses a reactive programming model, automatically updating the interface in response to changes in application state.
- **Backend Adapters**: Enable the generated code to integrate with different platforms, such as desktop, web, and embedded systems. These adapters ensure that the same interface definition can be rendered consistently regardless of the target environment.

Supported Platforms and Languages

Slint is designed with cross-platform compatibility in mind, making it suitable for a wide range of applications. It supports:

- **Desktop Environments**: GUIs can be rendered on desktop operating systems like Windows, macOS, and Linux.
- **Web Applications**: Through its JavaScript backend, Slint enables seamless deployment of GUIs in web browsers.
- **Embedded Systems**: Its lightweight design and minimal resource requirements make it particularly well-suited for embedded devices with constrained hardware.
- **Mobile Platforms**: While not currently supported, Slint has the potential to extend its reach to mobile platforms like Android and iOS.

In terms of programming languages, Slint primarily targets Rust[2], C++, and JavaScript, providing robust integration and efficient code generation for these ecosystems.

Strengths of Slint

One of the key strengths of Slint lies in its simplicity and accessibility. By using a DSL, it reduces the learning curve for developers who may not be familiar with low-level GUI programming. Additionally, the framework's declarative nature allows for a clear separation of concerns, making it easier to maintain and scale applications.

Another significant advantage is its cross-platform support. Developers can define a GUI once and deploy it across multiple platforms without significant modifications, ensuring consistency and reducing development time. This is particularly valuable in environments where applications need to operate on both desktop and embedded devices.

The reactive programming model is another standout feature. By automatically managing updates to the GUI in response to state changes, Slint reduces the need for manual event handling, which simplifies development and minimizes potential bugs.

[2] `https://www.rust-lang.org/`

Limitations of Slint

Despite its strengths, Slint has some limitations that restrict its applicability in certain scenarios.

One notable constraint is its limited language support. While it performs well in Rust, C++, and JavaScript, it does not natively support other widely used programming languages. Extending Slint to a new language is not straightforward due to its tightly integrated runtime and rendering engine, which are designed with specific language bindings in mind. Slint requires deep integration at the runtime level, making the process more complex and time-consuming.

Furthermore, while Slint's architecture supports modular backends, enabling developers to create custom platform abstractions and window adapters, this modularity is primarily focused on the rendering layer and platform-specific behaviors. The DSL itself remains monolithic in design, meaning that all features and components are bundled together, with no built-in mechanism to selectively enable or disable specific functionalities. This design choice increases the difficulty of porting Slint to new ecosystems, as it requires adapting not just the syntax and semantics but also the entire rendering pipeline to fit the constraints of the target language.

3.1.2 Glade: A GUI Designer for GNOME

Glade[3] is a versatile graphical user interface (GUI) designer primarily developed for building GTK-based applications. It enables developers to design interfaces visually, facilitating rapid prototyping and development. Part of the GNOME[4] ecosystem, Glade is widely adopted for creating applications that integrate seamlessly with GTK and the GNOME desktop environment.

Overview and Architecture

At its core, Glade follows a WYSIWYG (What You See Is What You Get) approach, allowing developers to construct interfaces by dragging and dropping widgets onto a visual canvas. This design paradigm simplifies GUI creation by abstracting the complexities of manual code writing, enabling developers to focus on the aesthetics and functionality of the interface.

Glade's design process centers on the XML-based UI description. Interfaces created with Glade are stored in `.ui` files, which are XML [10] representations of the GUI. These files can be loaded at runtime using GTK libraries, eliminating the need for code generation or manual implementation of the GUI layout. This decoupling of interface design from application logic promotes cleaner code organization and easier maintenance.

The tool integrates tightly with the GTK ecosystem, supporting an extensive array of widgets, properties, and signals. Developers can fine-tune widget behavior, layout,

[3] `https://glade.gnome.org/`
[4] `https://www.gnome.org/`

and appearance, ensuring flexibility and customization to meet specific application requirements.

Code Sketching in Glade

An important feature of Glade is its support for code sketching. Code sketchers are tools or extensions that generate source code from Glade's `.ui` files. Most commonly, these sketchers produce code that utilizes `libglade` and the `.ui` file to dynamically create the GUI at runtime. This approach maintains the separation of concerns between interface design and application logic.

Some code sketchers, however, go further by generating raw source code that does not depend on `libglade` or the original `.ui` file. These tools embed the interface logic directly into the application's source code, which may be beneficial in scenarios requiring standalone codebases without runtime dependencies. While this increases independence, it also ties the GUI more closely to the application, reducing flexibility for future modifications.

Supported Platforms

Glade and its GTK-generated GUIs are inherently cross-platform, capable of running on Linux, Windows, and macOS. However, Glade itself is most commonly used within Linux environments due to its integration with GNOME. The cross-platform nature of GTK ensures consistent behavior and appearance across supported platforms, making Glade a robust choice for developers targeting diverse deployment environments.

Strengths of Glade

Glade provides significant advantages for developers such as its visual design approach, which allows developers to construct user interfaces using a drag-and-drop interface.

Another key benefit of Glade lies in its XML-based workflow. The generation of `.ui` files to define the GUI separates the user interface from the core application logic. This separation promotes better organization, easier maintenance, and greater modularity, as developers can update or redesign the GUI without altering the underlying logic. This approach aligns with modern software engineering principles, such as separation of concerns, making it a valuable asset for maintaining code quality in larger projects.

The tool's support for code sketching adds another layer of flexibility. Developers can choose to generate either libglade-based code, which dynamically loads the `.ui` file at runtime, or raw source code that integrates the GUI directly into the application. This flexibility allows developers to adopt the approach that best fits their project requirements, balancing runtime modularity and standalone code generation as needed.

Limitations of Glade

Despite its strengths, Glade has limitations that can impact its effectiveness in certain scenarios. One of the most prominent drawbacks is its tight coupling with the GTK

framework. While GTK is powerful and versatile, this dependency restricts Glade's applicability to projects outside the GTK ecosystem. Developers working with other frameworks or technologies may find Glade unsuitable, requiring them to look for alternative tools.

Another limitation of Glade is its reliance on monolithic `.ui` files to define user interfaces. While these XML-based descriptions provide a structured way to represent GUIs, they can lack modularity. For instance, developers may struggle to selectively enable or disable specific interface components or features without modifying the entire `.ui` file. Although GTK allows loading separate `.ui` files dynamically, Glade itself does not provide built-in mechanisms for defining reusable templates or modular components that can be easily imported across different UI designs. This limitation can pose challenges for projects requiring highly customizable or dynamic UIs, as developers must manually structure their interface definitions and handle component reuse programmatically.

Finally, Glade's design and workflow are heavily optimized for GNOME environments, potentially resulting in less polished experiences on non-Linux platforms. Although GTK ensures cross-platform compatibility, developers targeting Windows or macOS may encounter inconsistencies or additional challenges when fine-tuning their applications for these environments.

3.1.3 Flutter: A Cross-Platform UI Framework

Flutter, developed by Google, is a popular open-source UI framework designed for creating natively compiled applications for mobile, web, and desktop from a single codebase. Since its initial release in 2017, Flutter has gained significant traction among developers due to its innovative approach to UI development and its ability to produce highly performant, visually consistent applications across platforms.

Overview and Architecture

Flutter is built on the Dart programming language[5], which plays a crucial role in enabling its features. The framework employs a unique rendering engine that bypasses traditional platform widgets, instead rendering the UI components directly on a canvas. This architecture allows Flutter to maintain consistent design and behavior across platforms while providing complete control over the application's appearance and performance.

The UI in Flutter is defined using a declarative approach, where developers describe the desired state of the interface and let the framework handle the rendering updates. This method simplifies the development process, particularly for complex and dynamic UIs. Flutter's rich set of pre-designed widgets caters to both material design and iOS-style applications, ensuring that developers can create visually appealing and platform-appropriate interfaces.

[5] `https://dart.dev/`

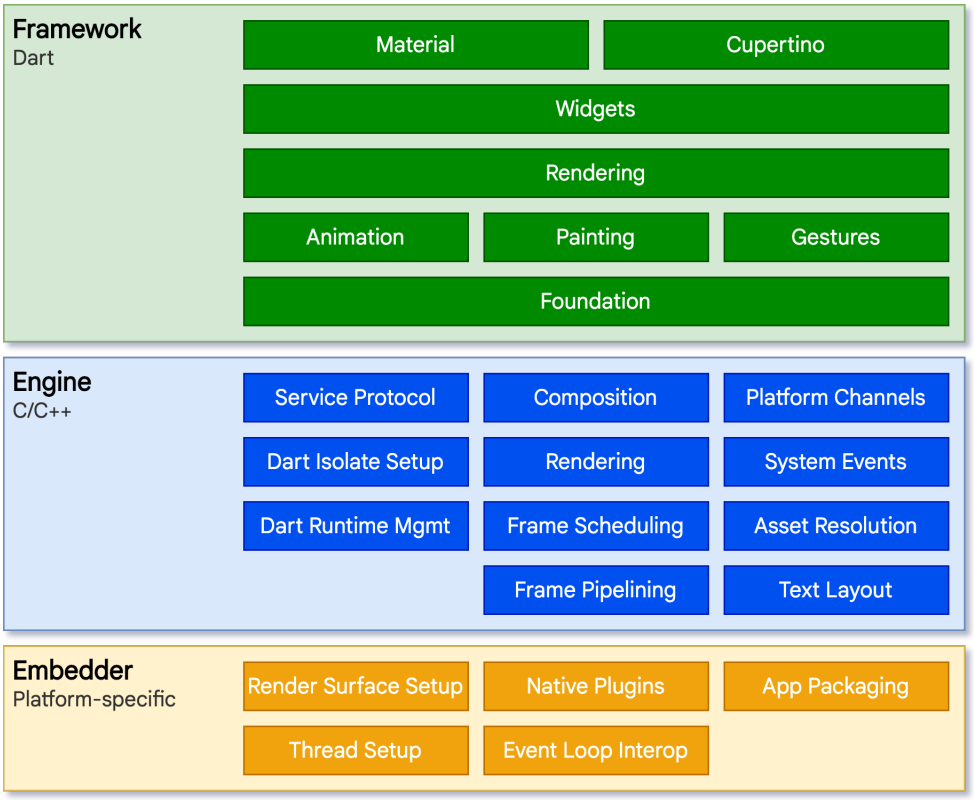

Figure 3.2. *Flutter's architecture is built around three primary layers: the Framework Layer, Engine Layer, and Embedder Layer. The framework provides foundational libraries and widgets, the engine handles rendering and platform interactions, and the embedder integrates Flutter with the underlying operating system .*

Flutter's architecture is built around three primary layers as shown in Figure 3.2:

- **Framework Layer**: This layer provides a collection of foundational libraries and pre-built widgets that developers use to design their interfaces. It includes support for animations, gestures, and platform-specific features.
- **Engine Layer**: The engine, written in C++, handles rendering, input, and platform-specific interactions. It uses Skia[6], a 2D graphics library, to draw the UI elements directly on the screen.
- **Embedder Layer**: This layer integrates the Flutter engine with the underlying operating system, enabling the application to run on various platforms, such as Android, iOS, Windows, macOS, Linux, and web browsers.

Strengths of Flutter

One of Flutter's standout strengths is its "write once, run anywhere" philosophy. Developers can use a single codebase to create applications that run on multiple

[6] `https://skia.org/`

platforms, reducing development time and effort significantly. The framework's hot reload feature further accelerates the development cycle by allowing developers to see changes in real-time without restarting the application.

Flutter's rendering engine enables high performance and pixel-perfect control over the UI, ensuring that applications look and behave consistently across platforms. This feature is particularly advantageous for developers targeting devices with varying screen sizes, resolutions, and operating systems.

Additionally, the declarative UI paradigm promotes clarity and maintainability in the codebase. Flutter's widget-centric approach makes it easy to break the UI into reusable components, enhancing both modularity and scalability.

Limitations of Flutter

Despite its advantages, Flutter has some limitations. The reliance on the Dart programming language can be a barrier for developers unfamiliar with it. While Dart is relatively easy to learn, its ecosystem and community are smaller compared to other mainstream languages, which may result in fewer resources and libraries.

Flutter's rendering engine, while powerful, increases the application's binary size, which can be a concern for mobile platforms with strict size limitations. Furthermore, the approach of bypassing native widgets means that Flutter apps may not feel entirely native to some users, as they do not use the system's built-in UI components.

Finally, while Flutter is cross-platform, achieving complete parity across all platforms requires additional effort. For example, some platform-specific features may need custom implementations using platform channels or third-party plugins.

3.2 Research Objectives

The existing technologies and frameworks discussed above provide valuable insights into the challenges and opportunities in graphical user interface (GUI) development and code generation. However, each solution has its limitations.

- **Glade**: While efficient for GTK applications, Glade lacks flexibility and modularity, relying heavily on monolithic `.ui` files and tightly coupling the tool to the GTK ecosystem. This restricts its applicability to other frameworks or platforms, making it difficult to extend or modify without altering the underlying code structure.
- **Slint**: Slint excels in supporting Rust, C++, and JavaScript but struggles with adaptability for other programming languages. Its DSL is monolithic, and it doesn't allow for selective modularity or customization, which limits its flexibility for developers working on diverse platforms or in different language ecosystems.
- **Flutter**: Although Flutter offers an attractive cross-platform development environment, it comes with its own set of challenges. It requires specific tools and dependencies, making it less suitable for developers who need to integrate with existing systems or work within specific language ecosystems.

These limitations highlight challenges in language support, modularity, and platform compatibility that impede the broader applicability of these tools. To address these shortcomings, this research aims to develop a novel approach that combines the strengths of domain-specific languages (DSLs) and software product lines (SPLs) to generate modular, language-agnostic GUI code tailored to specific project requirements.

The primary objectives of this research are as follows:

1. **Develop a Domain-Specific Language (DSL) for GUI Specification**: Create a DSL that allows developers to define GUIs using high-level abstractions, encapsulating common design patterns and interactions. The DSL should support declarative syntax, enabling developers to focus on the structure and behavior of the interface rather than low-level implementation details.
2. **Implement a Code Generation Framework for GUIs**: Develop a code generation framework that translates DSL specifications into executable code for various programming languages and platforms. The framework should support modular backends, enabling developers to target different environments without modifying the DSL definitions.
3. **Explore Software Product Line (SPL) Concepts for GUI Generation**: Investigate the application of SPL concepts to GUI generation, allowing developers to configure and customize the generated code based on specific project requirements. By leveraging SPL variability mechanisms, developers can create highly adaptable and reusable GUI components.

By achieving these objectives, this research aims to advance the state of the art in GUI development and code generation, providing developers with a flexible, efficient, and language-agnostic solution for creating modern graphical user interfaces.

4

Architecture

This chapter explore into the theoretical framework and architecture designed to address the challenges identified in the previous chapter. The proposed solution aims to streamline the generation of graphical user interfaces (GUIs) across multiple target languages while maintaining flexibility, modularity, and scalability. Central to the approach is an architecture composed of distinct layers, each serving a specific role within the system.

The architecture is structured to emphasize decoupling and modularity, ensuring that each layer can evolve independently while maintaining compatibility. At its core lies the Library Layer, which encapsulates the core logic, rules, and abstractions for constructing GUIs. This library, designed with a high degree of flexibility, can function independently of the Domain-Specific Language (DSL), offering the possibility of direct usage for more granular control.

The DSL Layer, on the other hand, provides a high-level abstraction that simplifies the creation of GUIs by allowing developers to compose interfaces using a concise, domain-specific syntax. The DSL leverages the foundational logic provided by the library to transform abstract specifications into tangible components. This design choice aims at hiding the complexity of GUI creation from the end user, promoting ease of use and efficiency.

To support multiple target languages, the architecture integrates adapters, which serve as connection between the core logic and the specific requirements of each language. This layer aims at ensuring that the generated code adheres to the conventions and practices of the respective language, guaranteeing compatibility and usability.

Finally, the entire system is designed with Software Product Line (SPL) principles, enabling the creation of modular versions of the application to specific use cases. Each module within the system can be selectively included or excluded based on the desired functionality.

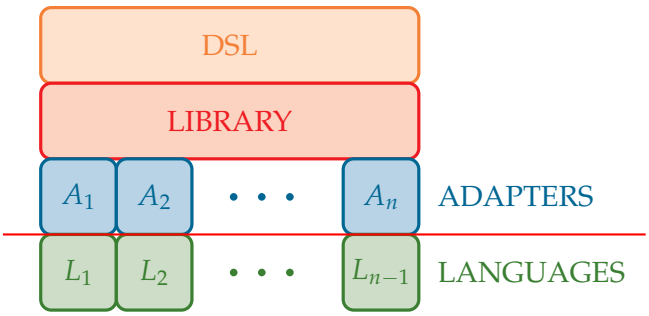

Figure 4.1. *The architecture of the system is organized into three primary layers: the DSL layer, which provides the language for GUI definitions; the Library layer, which houses abstractions and manages the logic for GUI element representation; and the Adapters layer, where specific code generators map the abstract definitions to target languages. Below the red line are the targeted programming languages, representing external systems outside the control of the developed solution. The modular design aims at extensibility and supports multiple target languages via dedicated adapters, enhancing flexibility and maintainability.*

This chapter is organized following the what's shown in Figure 4.1, with each section focusing on a specific layer of the architecture:

- **Library Layer**: This section explores the foundational abstractions and patterns used to construct GUIs, emphasizing the independence and flexibility of the library.
- **Adapter Layer**: The role of adapters in connecting the library with multiple languages and ensuring modularity will be discussed.
- **DSL Layer**: The functionality and purpose of the DSL, particularly its integration with the library, are presented here.
- **Software Product Line (SPL) Integration**: Finally, the SPL approach used to achieve modularity and feature management is explained.

By structuring the architecture in this way, the system aims to address the identified limitations of existing solutions, particularly the lack of modularity and flexibility, while offering a robust foundation for future extensions and integrations.

4.1 Library Layer

The Library Layer represents the foundation of the system's architecture, housing the essential logic, rules, and components necessary for constructing graphical user interfaces (GUIs). This layer is designed to function independently of the Domain-Specific Language (DSL), ensuring its versatility for various applications. By decoupling the library from the DSL, the architecture remains flexible and adaptable, enabling the library to be used programmatically or in combination with the DSL for enhanced abstraction.

At its core, the library is organized around the concept of abstract and concrete components. Abstract components define the general behavior and characteristics of

GUI elements, serving as templates or blueprints for their implementation. For example, a generic button may define attributes such as its label and the events it can handle, but leave the specifics of how these are rendered or executed to be implemented by concrete versions of the button.

Concrete components, on the other hand, provide the specific implementation details for each target programming language. For instance, the visual representation and behavior of a button in Python or Java may vary significantly, and concrete components handle these differences. This distinction between abstract and concrete components aims at keep the library extensible and adaptable to multiple languages without sacrificing consistency or functionality.

The library employs the Composite Design Pattern to manage the hierarchical structure of GUIs. Each GUI component can act as either a composite, capable of containing other components, or a leaf, which represents standalone elements. This pattern allows developers to construct complex GUIs by nesting smaller, reusable components within larger structures.

For instance, a container component like a panel or a window can act as a composite, holding child components such as buttons, text fields, or other containers. This hierarchical organization aims at keep the GUI structured and that rules governing component relationships are enforced at the library level. This approach enhances scalability, as developers can create intricate GUIs by composing simpler building blocks, and reusability, since components can be shared across different parts of the interface.

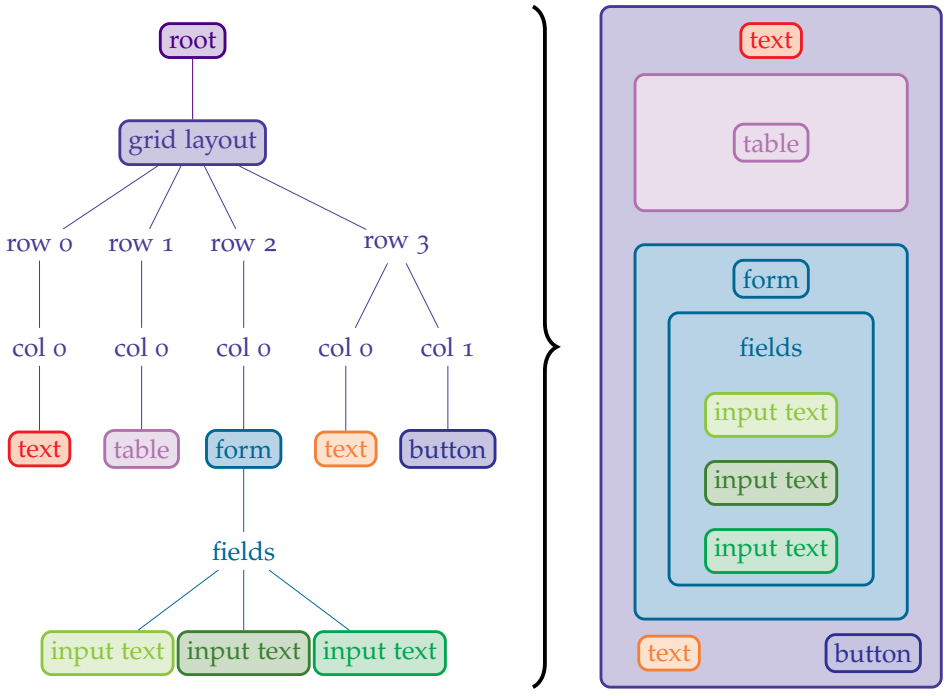

Figure 4.2. *An example illustrating the Composite Design Pattern in a GUI. The left side shows the abstract structure, while the right side displays the resulting visual layout.*

An example of this pattern is illustrated in Figure 4.2, which shows both the abstract structure of a GUI and its corresponding visual representation. In this example, a grid layout acts as a composite, organizing several rows and columns, each of which contains either other composite components (e.g., a form) or individual leaf components (e.g., text or button). This dual representation helps demonstrate how the logical structure of a GUI maps directly to its visual layout.

To manage the complexity of creating and configuring these components, the architecture employs the Builder Pattern. Each abstract component provides a dedicated builder class, which encapsulates the logic for incrementally setting fields and configuring properties. This design simplifies the process of creating instances of complex components, ensuring that they are always properly initialized before use.

A notable feature of the library is its complete independence from the DSL. While the DSL simplifies the process of creating GUIs by offering a high-level abstraction, the library itself can function autonomously. Developers have the option to directly use the library to construct GUIs without relying on the DSL. This design choice aims at keep the library flexible and can help to various use cases, including those where the DSL might not be suitable or required.

Moreover, the independence of the library allows it to be easily integrated with alternative frameworks or tools in the future. By isolating the core functionality of GUI creation within the library, the system avoids tying itself to a specific DSL, enhancing its longevity and adaptability.

The library is inherently modular, with each component implemented as a separate, self-contained module. This modularity is in line with the principles of Software Product Lines (SPLs), enabling the creation of customized versions of the library that include only the necessary features and components. For example, a deployment targeting a specific platform or language might only include the relevant concrete components, reducing overhead and ensuring efficiency.

Additionally, the modular design supports scalability by allowing new components to be added without disrupting the existing structure. This is particularly beneficial as new requirements or target languages emerge, ensuring that the library can evolve alongside changing needs.

In conclusion, the Library Layer is the cornerstone of the system's architecture. Its robust, modular design aims at flexibility and adaptability, while the use of patterns like Composite provides a structured and reusable approach to GUI construction. By remaining independent of the DSL, the library serves as a versatile and extensible foundation for the entire system.

4.2 Adapter Layer

The adapter layer serves as a critical intermediary between the core logic defined in the library and the specific requirements of the target programming languages. Its primary role is to aim at keep the abstract components and structures defined in the library are correctly translated into language-specific implementations. By encapsulating

these translations, adapters decouple the library's core logic from the complexities of individual languages, thus promoting flexibility and reusability.

In this architecture, adapters act as language-specific translators. Each adapter is tailored to a particular target language, such as Python, JavaScript, or C++, and defines the mechanisms required to generate code that aligns with the conventions, syntax, and idioms of that language. This design choice aims at the integration of the generated GUIs into existing codebases seamlessly, following the best practices of the target platform.

One of the key strengths of the adapter layer is its modularity. Each adapter operates independently of the others, meaning that the system can support multiple target languages without requiring changes to the library or other adapters. This modularity also simplifies the process of adding support for new languages. To include a new language, developers need only implement a new adapter that adheres to the predefined interfaces established by the library. This approach minimizes the risk of introducing errors or dependencies that could compromise the stability of the overall system.

Moreover, the adapter layer aims at create an architecture that is robust in handling variations between languages. For instance, certain GUI frameworks or programming languages may impose specific constraints or offer unique features that must be accommodated during code generation. Adapters handle these variations by customizing the way components are instantiated, configured, and rendered. This abstraction not only simplifies the development process but also aims at geenerating GUIs that are consistent across different platforms.

Another crucial element of the architecture is the use of factories for generating concrete components to specific target languages. Each factory is dedicated to a single programming language and contains methods designed to create various UI components, such as buttons, containers, or text fields, according to the syntax and conventions of the respective language.

When a user specifies a target language, the system dynamically selects the appropriate factory. This factory then generates the required components, which are subsequently assembled into the final UI structure using the composite pattern. This aims at creating an approach for code generation that is both modular and scalable.

For example, if the target language is Java, the corresponding factory would generate UI elements compatible with Java-based GUI frameworks like JavaFX. Similarly, a factory for Python will generate code for Python-based GUI frameworks such as Tkinter[1]. By isolating the generation logic within language-specific factories, the library achieves a high degree of separation of concerns, making it easier to add support for new languages in the future without disrupting existing functionality.

By isolating language-specific concerns within the adapter layer, the architecture adheres to the principles of separation of concerns and single responsibility. The library focuses solely on defining the core logic and structure of GUIs, while adapters handle the task of translating this logic into concrete implementations. This separation enhances maintainability, as updates or modifications to the library do not require

[1] https://docs.python.org/3/library/tkinter.html

changes to the adapters, and vice versa.

In summary, the adapter layer plays a pivotal role in closing the gap between the abstract, language-agnostic logic of the library and the concrete, language-specific implementations required for GUI generation. Its modular and extensible design aims at creating an architecture that the can evolve to support new languages and frameworks while maintaining the integrity and reusability of its core components.

4.3 DSL Layer

The DSL in this architecture serves as a high-level, human-readable interface for defining the structure and behavior of graphical user interfaces (GUIs). While the library encapsulates the logic and components necessary for GUI generation, the DSL abstracts this complexity, offering users a simplified and intuitive way to specify their desired GUI elements and configurations.

At its core, the DSL allows users to describe the hierarchical structure of a GUI in a manner similar to constructing a document object model[2] (DOM). Each element in the DSL corresponds to a component defined in the library, such as buttons, containers, or input fields. By leveraging the composite design pattern, the DSL aims to support the nesting and combination of components in ways that align with the intended behavior of the GUI. In addition, the DSL integrates the Builder Pattern from the abstract components of the library to configure the fields required by the DSL code written by the user.

The separation between the DSL and the library is a deliberate design choice aimed at promoting modularity and adaptability. The library operates independently of the DSL, defining all GUI components and their interactions. This independence aims to allows the library to be used on its own, without requiring the DSL. For example, developers could directly instantiate and configure library components programmatically if desired. However, such an approach would be labor-intensive and prone to errors, particularly for complex GUIs.

The DSL, on the other hand, simplifies the use of the library by providing a higher-level syntax for describing GUIs. Users can define what they want their interface to look like and how it should behave without delving into the details of instantiating and configuring components manually. The DSL effectively reduces the barrier to entry for developers who may not have extensive experience with the underlying library, enabling a wider range of users to leverage its capabilities.

A significant advantage of this design is the potential to decouple the DSL from the implementation. Since the library is not dependent on the DSL, it is possible to replace or modify the DSL in the future without requiring changes to the library. This decoupling promotes long-term flexibility, allowing the architecture to adapt to new paradigms or tools for defining GUIs.

The DSL also integrates closely with the modular design enabled by the software product line (SPL). Each component of the DSL is associated with a corresponding

[2] https://dom.spec.whatwg.org/

module in the library, ensuring that the DSL can only reference components that are included in the current configuration. This alignment between the DSL and the SPL helps maintain consistency between the DSL and the library's capabilities, reducing the risk of errors caused by referencing unavailable components.

In summary, the DSL serves as a user-friendly interface for leveraging the power of the library in defining GUIs. Its abstraction reduces complexity, streamlines the development process, and enhances accessibility for developers. By maintaining independence from the library and aligning with the SPL, the DSL supports flexibility, modularity, and long-term adaptability, making it a crucial component of the overall architecture.

4.4 Software Product Line (SPL) Integration

The Software Product Line (SPL) approach is a cornerstone of the architecture, enabling the system to achieve high levels of modularity, flexibility, and configurability. SPL is particularly well-suited to the requirements of this project, as it allows for the creation of versions of the software by assembling only the components necessary for specific use cases.

At its core, the SPL organizes the system into distinct modules, each representing a specific feature or functionality. In this architecture, these modules correspond to the various components of the library and DSL. For instance, there are modules for abstract library components, concrete implementations for specific languages, and the grammar definitions of the DSL. By isolating these elements into self-contained units, the SPL makes it possible to customize the system dynamically.

The modularity introduced by SPL promotes a highly adaptable architecture. Developers can selectively include or exclude modules depending on the requirements of a particular use case. For example, if a project targets only a specific GUI language, such as Flutter, the SPL can produce a configuration containing only the Flutter-related modules, omitting unnecessary elements. This not only reduces the footprint of the resulting application but also simplifies its structure, making it easier to maintain and debug.

Another advantage of this modular approach is the ability to scale the system. As new GUI languages or frameworks emerge, additional modules can be created and seamlessly integrated into the SPL. This extensibility helps keep the system relevant and capable of accommodating evolving requirements without requiring significant architectural changes.

The SPL aims at creating a consistent relationship between the DSL and the library by tightly controlling the components available in a given configuration. The DSL grammar is designed to correspond directly to the modules included in the library. For instance, if a particular button or container is part of the library configuration, the DSL grammar will support its definition. Conversely, if a component is excluded from the library, the DSL will not generate references to it. This alignment eliminates potential mismatches, ensuring that the DSL remains valid and functional for the current configuration.

One of the primary benefits of the SPL approach is its ability to address the challenge of feature variability. Traditional monolithic systems often include all possible features, leading to bloated and inefficient implementations. In contrast, the SPL approach allows for lean configurations to specific requirements, reducing resource consumption and improving performance.

The SPL also enhances maintainability. Since each module is self-contained, updates or bug fixes can be applied in isolation, reducing the risk of unintended side effects. Moreover, the modular design simplifies testing, as individual modules can be verified independently before being integrated into the larger system.

In practice, the SPL enables users of the system to generate customized versions of the software that align with their specific needs. This could involve selecting only the necessary GUI components, supporting only the desired target languages, or enabling specific features of the DSL. The SPL handles the complex task of assembling these elements into a cohesive and functional system, allowing users to focus on their application rather than the underlying infrastructure.

A key aspect of this process is the way configurations are used to collect only the necessary dependencies and generate an optimized variant of the DSL. Instead of including all possible features, which could lead to unnecessary complexity and increased resource consumption, the SPL assembles a streamlined version that contains only the selected functionalities. This prevents the DSL from becoming bloated with unused features and ensures that the final system remains efficient and maintainable. The collection process relies on predefined configuration rules and dependency resolution mechanisms, which determine how individual modules interact and ensure that all required components are included while excluding unnecessary ones.

Overall, the SPL is a critical enabler of the architecture's goals. By providing modularity, configurability, and scalability, it supports the creation of efficient, targeted solutions while maintaining a high degree of adaptability. This aims to create a system that remains robust, future-proof, and capable of meeting a diverse range of user requirements.

5 Implementation

This chapter digs into the practical realization of the architecture outlined in the previous section. While the design provides a high-level perspective of the system's structure and interactions, the implementation details reveal how these concepts are translated into a working system. The primary objective of this chapter is to illustrate how the theoretical components, such as the library, DSL, adapters, and SPL module, are brought to life through code, tools, and frameworks.

The implementation process emphasizes modularity and maintainability, reflecting the core principles of the system's design. Each module of the project, ranging from the abstract library and DSL to the SPL utilities, is implemented as a standalone unit with clear interfaces and responsibilities. This modularity not only simplifies development and testing but also supports future extensibility, such as adding new languages or features.

The chapter is organized to follow the system's core modules, discussing the implementation details of each. Key topics include the structure and functionality of the library, the DSL grammar and parser, the adapter mechanisms for language-specific code generation, and the tools provided by the SPL module. Additionally, this chapter highlights the use of programming patterns, such as the composite and factory patterns, and the strategies employed to ensure compatibility across supported platforms.

By exploring these aspects, this chapter demonstrates how the theoretical concepts are realized in code and how the overall system achieves its goal of facilitating modular and feature-rich UI code generation.

5.1 Technologies Used

The development of this project is supported by a carefully selected set of technologies, chosen for their ability to facilitate modularity, maintainability, and extensibility. The main technologies employed include Java 17, Neverlang, and Gradle. Java 17 serves as the primary programming language for the project. Its robust ecosystem, object-oriented paradigm, and extensive library support make it a versatile choice for building complex systems. Java also provides strong support for implementing design patterns, such as the Composite Pattern and Abstract Factory Pattern, which are crucial for the modular architecture of the project. Additionally, its platform independence ensures that the project can be executed across different environments without requiring significant changes.

Neverlang is used as the foundation for developing the domain-specific language (DSL). This language workbench simplifies the creation of DSLs by providing tools for defining grammar, syntax, and semantics in a modular manner. The modularity of Neverlang aligns well with the overall design philosophy of the project, allowing components to be developed independently and integrated seamlessly.

Gradle is employed as the build automation tool, providing a structured approach to managing the project's modular organization. Its flexibility allows for an efficient build process, with each folder in the project corresponding to a distinct module. This modular setup ensures clear separation of concerns, streamlines the development process, and simplifies maintenance. Gradle's ability to manage dependencies and configurations efficiently is instrumental in supporting the project's dynamic nature.

Together, these technologies form the foundation of the project, enabling the development of a modular and extensible system while maintaining clarity and efficiency throughout the implementation process.

5.1.1 Target Languages

The system is designed to generate user interfaces in multiple programming languages, each tailored to different environments and use cases. The three target languages currently supported are HTML, Python (Tkinter), and Elixir (Phoenix LiveView). Each of these languages follows a distinct paradigm and requires different handling when generating UI components.

The three supported target languages represent distinct paradigms: HTML provides a simple, standalone web UI; Python (Tkinter) generates a desktop GUI with a hierarchical component structure; and Elixir (Phoenix LiveView) integrates into an existing framework, enabling dynamic, real-time web applications. This variety allows users to generate UI code suited to their specific deployment needs, whether for a simple webpage, a local desktop application, or a complex web system.

HTML (Bootstrap-based)

The HTML output consists of a standalone file that includes the entire UI structure. The generated HTML uses Bootstrap for styling and incorporates minimal JavaScript to handle user interactions. The JavaScript primarily enables event callbacks, allowing UI components to invoke user-defined functions. Since HTML is inherently web-based, the generated files can be deployed and accessed in any modern browser without additional setup.

Python (Tkinter)

Unlike HTML, the Python output is intended for desktop applications. The generated code uses Tkinter, a lightweight GUI toolkit included with Python. Since Tkinter requires a hierarchical structure where each component belongs to a parent (or "master"), the system ensures that layouts correctly pass down the required references to their

child components. This makes the Python-generated UI fundamentally different from the web-based approach, as it directly integrates with the host system instead of running in a browser.

Elixir (Phoenix LiveView)

The Elixir output targets Phoenix LiveView, a modern web framework that enables real-time, server-rendered applications. Unlike the standalone nature of HTML output, Phoenix LiveView requires the generated UI code to be integrated within an existing project structure. The UI elements are designed to work within Phoenix templates, following the conventions of LiveView components. The system also uses Tailwind CSS for styling, ensuring a modern and responsive design. Because LiveView dynamically updates the UI based on server-side events, this approach is more interactive than static HTML while still being web-based.

5.2 Project Structure

The project is organized into a modular architecture that separates different functionalities into distinct folders, ensuring scalability, maintainability, and clarity. This structure reflects the conceptual layers of the system and allows for targeted development and testing. Below is an overview of the main components of the project structure:

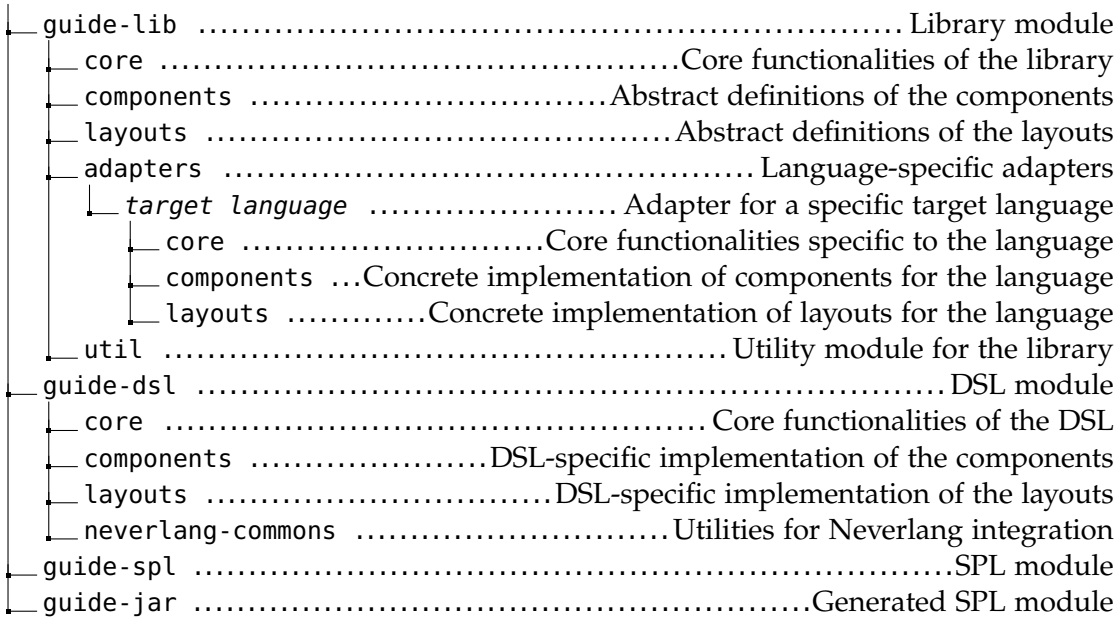

Figure 5.1. *Overview of the project folder structure, highlighting the modular organization of the library, DSL, SPL.*

5.2.1 `guide-lib`: Library Module

The `guide-lib` folder contains the core functionalities and abstract definitions that form the backbone of the application. This module is responsible for defining the structure and behavior of the library components, which can be utilized independently of the DSL. Key subfolders include:

- **`core`**: Houses the essential utilities and functionality required by the library.
- **`components`**: Provides abstract definitions for the GUI components, such as buttons, text fields, and other interface elements.
- **`layouts`**: Contains abstract definitions for layout structures, specifying how components are organized spatially.
- **`adapters`**: Implements language-specific variations of the library using modular adapters. Each adapter folder contains submodules for:
  - `core`: Core functionality specific to the target language.
  - `components`: Concrete implementations of the GUI components for the language.
  - `layouts`: Concrete implementations of the layout structures for the language.
- **`util`**: Includes utility classes and functions shared across the library.

5.2.2 `guide-dsl`: DSL Module

The `guide-dsl` folder defines the domain-specific language (DSL) used to describe the desired GUI. This module provides a high-level abstraction over the library to simplify the creation of user interfaces. Its submodules include:

- **`core`**: Implements the core grammar and parsing logic for the DSL.
- **`components`**: Contains the DSL-specific logic for mapping GUI components to the library.
- **`layouts`**: Provides DSL-specific logic for defining layouts.
- **`neverlang-commons`**: Includes utilities for integrating the DSL with the Neverlang framework, enabling modular language composition and extension.

5.2.3 `guide-spl`: SPL Module

The `guide-spl` module provides two main functionalities essential for managing the Software Product Line (SPL) of the project:

- **Feature-Based Composition**: This functionality allows developers to generate the `guide-jar` module by selecting a list of desired features. The program takes these features as input and creates a version of the library and DSL with only the necessary parts, enabling the creation of a tailored deployable product.
- **Utility Programs for Extension**: The module includes utility programs designed to assist developers in extending the system. These tools automate the creation of new components, layouts, or language adapters. Once run, the utilities generate

the required files and folder structures in their appropriate locations, including preliminary definitions without implementations. This simplifies the process of adding new features while maintaining consistency across the project.

5.2.4 `guide-jar`: Generated SPL Module

The `guide-jar` folder contains the output of the SPL process, producing a module that is immediately ready for the release process without requiring any manual modifications. Once the release process is completed, the resulting product is fully prepared for use by developers utilizing the DSL to create graphical user interfaces. This approach ensures a streamlined workflow, allowing users to work with a customized and optimized version of the library and DSL based on the selected features.

5.3 Library Implementation

The library module (`guide-lib`) is the core component of the system, providing the foundational building blocks for creating user interfaces. This section centers around the implementation details of the library, focusing on the structure and functionality of the core, components, layouts, and adapters.

5.3.1 Core

The Component Abstract Class

At the heart of the core module lies the `Component` class (Listing 5.1), which serves as the foundational abstraction for all UI elements. It is an abstract class designed to encapsulate the essential properties and behaviors of a user interface component. A special case of `Component` is `Layout`, another abstract class that extends it, specifically used to distinguish structural elements from standard components.

Each `Component` instance is uniquely identified by an ID and maintains a set of attributes, managed through a key-value map. Initially, this map is empty, but attributes can be dynamically set during the rendering process. This mechanism allows components to pass relevant contextual information to their children. For instance, in Python's Tkinter, each component requires a `master` element. When rendering a layout, it assigns the `master` attribute to its child components, ensuring that they automatically receive the correct parent reference when their `render` method is invoked.

The `Component` class also employs the Builder Pattern to streamline the configuration of its attributes. This pattern enables developers to incrementally set properties of a component.

The core functionality of `Component` is embodied in its abstract `render` method. This method is responsible for producing the corresponding source code representation of the component. When invoked, it returns a string containing the generated code, which can either be written to a file or used by another component as part of a larger structure. This mechanism is central to the composite pattern, allowing components to be nested

and dynamically combined to form complex UI hierarchies while maintaining a clear structure and attribute propagation.

```
public abstract class Component {
    private static int id_number = 0;
    private Value<String> id;
    private final Map<Attributes, String> attributes = new HashMap<>();
    public Component() {
        id = Value.of(this.getClass().getSimpleName() + id_number++);
    }
    public abstract String render();
    public String id() {
        return id.get();
    }
    public Component withId(Value<String> id) {
        this.id = id;
        return this;
    }
    public void setAttribute(Attributes key, String value) {
        attributes.put(key, value);
    }
    public Optional<String> getAttribute(Attributes key) {
        return Optional.ofNullable(attributes.get(key));
    }
}
```

Listing 5.1. *Implementation of the `Component` abstract class .*

Callback Handling

Another essential part of the core library is the `Callback` class (implemented as a `record` in Listing 5.2), which represents an event-driven action within the UI. A callback can be assigned as a parameter to components that require user interaction, such as a button's click event or an input field's change event.

```
public record Callback(String name) {}
```

Listing 5.2. *Implementation of the `Callback` record, which represents a reference to a user-defined function invoked by UI components. In Java, a `record` is a special type of class designed to concisely model immutable data carriers, automatically providing constructor, accessors, `equals()`, `hashCode()`, and `toString()` methods .*

Each `Callback` instance stores the name of the function that should be executed in the target language when the event is triggered. This allows the generated UI code to properly reference and invoke user-defined functions within the specific execution environment. The callback mechanism ensures that interactive elements are not only visually represented but also functionally integrated into the application logic.

Content Abstraction

The `Content` class (Listing 5.3) is another fundamental part of the core library, designed to encapsulate different types of content that can be included within components. Rather than limiting component content to a single data type, this abstraction allows greater flexibility in UI composition.

A `Content` instance can represent:

- A hardcoded string, which directly defines static content.
- A `Callback`, enabling dynamic content retrieval through a function call in the target language.
- Another `Component`, allowing the nesting of UI elements (e.g., embedding a button inside a table cell).

This abstraction is particularly useful for complex components such as tables, where each cell can contain static text, dynamic content generated at runtime, or even interactive elements. By leveraging the `Content` class, the library ensures a modular and extensible approach to UI representation.

```
public class Content<T> {
    private ContentType type;
    private Value<T> value = Value.of(() -> null);

    public Content() {}

    public Content(ContentType contentType, Value<T> value) { ... }

    public static Content<?> contentString(String value) {
        return new Content<>(ContentType.STRING, Value.of(value));
    }

    public static Content<?> contentCallback(Value<Callback> value) {
        return new Content<>(ContentType.CALLBACK, value);
    }

    public static Content<?> contentComponent(Value<Component> value) {
        return new Content<>(ContentType.COMPONENT, value);
    }

    public ContentType type() { return type; }
    public T value() { return value.get(); }

    public Content<T> withType(ContentType type) { ... }
    public Content<T> withValue(Value<T> value) { ... }

    public enum ContentType {
        STRING, CALLBACK, COMPONENT
    }
}
```

Listing 5.3. *Implementation of the `Content` class, representing different types of values (strings, callbacks, or components) that can be used within a UI component .*

Lazy Evaluation with Value

The `Value` class (Listing 5.4) serves as a wrapper to enable lazy evaluation within the library. This design follows the Hollywood Principle (*"Don't call us, we'll call you"*), ensuring that computations and function calls within the component tree, built using the Composite Pattern, are only executed during the rendering phase.

Instead of immediately resolving values when a component is instantiated, `Value` delays evaluation until the `render` function is invoked. This mechanism improves efficiency by preventing unnecessary computations and allows components to dynamically adjust their output based on the attributes and state of their parent elements.

By incorporating `Value`, the library maintains a clean separation between component definition and execution, optimizing performance while maintaining flexibility in UI generation.

```
public class Value<T> {
    private final Supplier<T> valueSupplier;
    public Value(T value) {
        this.valueSupplier = () -> value;
    }
    public Value(Supplier<T> supplier) {
        this.valueSupplier = supplier;
    }
    public T get() {
        T value = valueSupplier.get();
        if (value instanceof Value) {
            return ((Value<T>) value).get();
        } else {
            return value;
        }
    }
    public static <T> Value<T> of(T value) {
        return new Value<>(value);
    }
    public static <T> Value<T> of(Supplier<T> supplier) {
        return new Value<>(supplier);
    }
}
```

Listing 5.4. *Simplified implementation of the `Value` class, which wraps a value or a supplier to support lazy evaluation. In Java, a `Supplier<T>` is a functional interface that provides a value on demand, allowing deferred computation and reducing unnecessary evaluations .*

5.3.2 Components and Layouts

The `components` and `layouts` directories define the concrete building blocks for constructing user interfaces. Although they include a Gradle build file to manage dependencies collectively, they are not monolithic modules. Instead, each individual

component and layout exists as a separate Gradle submodule. This modular structure allows the Software Product Line (SPL) to selectively include only the necessary components during the generation process, avoiding unnecessary dependencies and reducing the final artifact size.

To facilitate the integration of components and layouts with different target languages, each component and layout has a corresponding factory interface. These interfaces define the methods required to instantiate concrete implementations of the components.

Each adapter implements these interfaces by providing language-specific factories that return the appropriate component instances. This design allows the SPL process to dynamically generate the necessary factories based on the selected features and target language.

By using factory interfaces, the architecture maintains a clear separation between abstract definitions and concrete implementations, ensuring that the core library remains independent of any specific language while allowing for seamless extensibility.

Components

The set of implemented components includes:

- `Root`: The `Root` component serves as the entry point for the rendering process. It is the only component that cannot be nested inside another and acts as the top-level container for the generated UI. The output of its `render` function represents the complete UI structure, which is then written to a file in the target language.

  The `Root` component has several key parameters:

  - `callbacksFilename`: Determines the name of the user-implemented callback file to be imported, allowing the generated UI to correctly reference external function definitions.
  - `layout`: Defines the layout structure used within the UI.
  - `module`: Specifies the name of the generated class in Python or module in Elixir.
  - `title`: Represents the UI title, which is displayed as the page title for web applications or as the window title for desktop applications.
  - `width` and `height`: Define the dimensions of the window in desktop environments.

  Since the `Root` component encapsulates the entire UI, it plays a central role in integrating all other components, ensuring they are structured correctly according to the selected target language.
- `Text`: A simple text element.
- `Button`: A clickable button that triggers callbacks.
- `Table`: A tabular structure for displaying data in rows and columns.
- `Form`: A container for grouping input elements.
- `InputText`: A text input field for user input.

Each component is implemented following the Composite Pattern, allowing hierarchical structuring where applicable. Components rely on the `render` function to generate the corresponding target language code. Additionally, certain components utilize inherited attributes propagated from their parent to determine configuration settings dynamically.

Layouts

Layouts define the positioning and structure of components within the UI. Unlike standard components, layouts do not represent standalone visual elements but rather describe how child components are arranged. Layouts are designed to be implemented as separate Gradle submodules, enabling selective inclusion during SPL-based generation. This modular approach allows future extensions by adding new layout types without affecting existing implementations.

Currently, the only implemented layout is the `GridLayout`, a layout system that arranges components in a flexible grid structure. The `GridLayout` was chosen as the initial implementation due to its versatility and widespread adoption in UI frameworks. Grid-based layouts provide a balanced trade-off between flexibility and simplicity, supporting various UI configurations by defining rows and columns. This makes them suitable for a wide range of applications, from simple forms to more complex dashboards. Additionally, the grid structure simplifies component alignment and resizing, which are essential features when generating GUIs for multiple target languages with differing layout management requirements.

5.4 Adapters Implementation

The adapters play a crucial role in the architecture, as they provide the concrete implementations of components and layouts for different target languages. These adapters are located within the `guide-lib` module, inside the `adapters` directory. This directory serves primarily as an organizational structure rather than a standalone module, similar to how the `components` directory is structured.

5.4.1 Structure of the Adapters

Each supported target language has its own subdirectory within `adapters`, which contains the necessary elements to generate the UI in that specific language. The structure of each language adapter is divided into three main sections:

- **Core Module**: The `core` directory contains language-specific utility classes that assist in generating UI code. Some languages may not require additional utilities, so their core module might be minimal or even empty.
- **Components and Layouts**: The `components` and `layouts` directories house the concrete implementations of UI elements for a given language. Each component

and layout extends its corresponding abstract definition from `guide-lib`, ensuring consistency while allowing for language-specific adaptations.

Each component or layout implementation follows a structured dependency model:

- It depends on the `core` module of its respective language for any required utilities.
- It depends on the `util` module of `guide-lib`, which provides shared functionalities across all adapters.
- It extends the corresponding abstract class from `guide-lib`, ensuring that all components adhere to a consistent interface.

5.4.2 Rendering and Code Generation

The primary responsibility of each adapter is to implement the `render` method for its components and layouts. This method generates the appropriate source code in the target language, ensuring that the final output aligns with the expected format.

To maintain readability and ease of maintenance, the implementation avoids embedding large code snippets directly as Java strings. Instead, each adapter relies on template files stored in the `resources` directory. These templates define the overall structure of the generated code while using placeholders for dynamic content.

When a component is rendered, the system retrieves the relevant template file and replaces placeholders with the actual values required to construct the final UI code. This approach not only improves maintainability but also enhances scalability, as modifications to the code structure can be made directly in the template files without altering the Java implementation.

The functionality for loading and processing these template files is provided by the `util` module in `guide-lib`. This ensures a clean separation between logic and presentation, making it easier to extend the system with additional target languages in the future.

5.5 DSL Implementation

The Domain-Specific Language (DSL) module, named `guide-dsl`, is structured similarly to `guide-lib`, following a modular approach that ensures flexibility and maintainability. It is divided into several key directories: `core`, `components`, `layouts`, and `neverlang-commons`. Each of these plays a distinct role in defining how the DSL functions and interacts with the underlying library.

5.5.1 Core

The `core` module is fundamental to the DSL, as it provides the essential functionalities required for any version of the product generated by the SPL. This module primarily focuses on defining the syntax and semantics of the DSL through a series of `Neverlang` modules. These modules establish the structural rules and behaviors that govern how user-defined programs are interpreted and transformed into executable UI code.

To achieve this, the `core` module depends on several key components:

- `guide-dsl:neverlang-commons` which provides utilities for error management, expressions, and type handling.
- `guide-lib:core` which includes the foundational abstractions shared across the entire library.
- `guide-lib:util` offering utility classes and functions used throughout the system.
- `guide-lib:components:root` which defines the root component structure critical for rendering and hierarchy management in the DSL.

In the `guide-dsl` core, the concepts of program, module, and variable form the foundation of the DSL's design and functionality.

```
1  Module mainModule {
2    // Variables
3    stringVariable = "Hello world"
4    intVariable = 42
5    boolVariable = true
6    colorVariable = #color("red")
8    /* Root definition */
9    myFirstRoot = Root {
10     layout: layoutModule.myFirstLayout
11   }
12 }
14 Module layoutModule {
15   myFirstLayout = GridLayout {}
16 }
```

Listing 5.5. *Example of a GUIDE DSL program defining modules and variables. The `mainModule` declares multiple variables of different types (string, integer, boolean and color) and instantiates a `Root` component using a layout defined in `layoutModule` .*

A GUIDE Program is defined as a collection of modules. These modules can be declared within the same file as shown in line 1 and lines 14 of Listing 5.5 or across multiple `.guide` files, which are automatically loaded when the GUIDE Program is executed. This behavior is analogous to the Java `classpath`, where all `.class` files within a directory are loaded into the execution environment. This mechanism ensures that all necessary modules and their dependencies are available for program execution.

Within each module, developers can define variables, as demonstrated in lines 3–6 of Listing 5.5. These variables serve as the core entities in the DSL and can represent various types, including components, layouts, colors, strings, numbers, or booleans. Modules are not isolated; the DSL supports cross-module referencing through dot notation (e.g., `layoutModule.myFirstLayout` in line 10), allowing variables from one module to be accessed and utilized within another. This capability is crucial for enabling modular and reusable designs within the DSL.

To execute a GUIDE Program, the user must specify the name of a variable designated as the `Root` of the program. In Listing 5.5, this is demonstrated in line 9, where the

`myFirstRoot` variable is defined as an instance of `Root`. This variable represents the starting point of the UI structure. The DSL uses this `Root` to generate the corresponding code, ensuring that the hierarchical relationships and dependencies among components are respected and accurately reflected in the output.

To support the functionality and usability of the DSL, the `core` module also defines foundational constructs such as comments, expressions, and identifiers.

The DSL allows developers to include comments within their `.guide` files to document their code and improve readability, as seen in line 2 with a line comment and line 8 with a block comment. Comments are ignored during execution and do not impact the generated output. This feature enhances the clarity of DSL programs, particularly in complex designs involving multiple modules and variables.

In the DSL, everything is treated as an expression, including the values assigned to variables (lines 3–6) and the parameters defined for components (e.g., `layout: layoutModule.layoutVariable` in line 10). This design choice ensures that the DSL remains highly flexible and consistent. By treating all constructs as expressions, the DSL can seamlessly handle both simple values (e.g., strings, numbers, or booleans) and more complex constructs, such as references to other variables or dynamically evaluated parameters.

To support this level of expressiveness, the DSL adopts the Hollywood Principle combined with lazy evaluation. This means that expressions are not immediately resolved when defined; instead, they are evaluated only when needed during execution. This approach ensures that components and parameters are resolved in the correct order, even in scenarios involving circular dependencies or deferred computations. It also allows users to define dynamic relationships between components without requiring them to manually manage evaluation timing or dependencies.

The DSL employs identifiers to uniquely name variables and modules. Identifiers are critical for ensuring that each variable or module can be distinctly referenced, whether within the same module or across modules via dot notation (e.g., line 10).

5.5.2 Components and Layouts

The `components` and `layouts` directories structure the DSL into Gradle submodules, each representing a specific GUI component or layout. This modular design aligns with the Software Product Line (SPL) approach, allowing selective inclusion of features based on project requirements.

Each submodule follows a consistent structure and depends on its corresponding abstract counterpart in the library. Specifically, every component or layout module in the DSL depends on its equivalent abstraction in `guide-lib`, ensuring a clear separation between the DSL syntax and the underlying implementation logic. Furthermore, all component and layout modules inherit dependencies on both the `core` module of the DSL and the `core` of the library, providing access to fundamental constructs necessary for correct integration and execution.

Within each submodule, a `Neverlang` module defines the syntax and semantics of the respective component or layout. These definitions specify how the element is parsed

within the DSL and how it translates into the corresponding library representation. The semantic implementation relies on the appropriate factory of the target output language, which is correctly passed by `Neverlang`. Once the base component is generated by the factory, the required fields specified in the DSL further customize it using the builder pattern, ensuring flexibility in adapting components to user-defined configurations.

5.5.3 Neverlang Commons

The `neverlang-commons` module contains modified versions of several libraries from the `Neverlang` framework to meet the requirements of `guide`. These libraries provide essential functionality for error management, expression evaluation, and type handling within the DSL.

The `errors` library is responsible for managing runtime errors that may occur during the execution of the DSL. Unlike syntax errors, which are handled during parsing, runtime errors are addressed through mechanisms provided by this library. These include detailed error reporting and recovery strategies to ensure the robustness of the DSL execution process.

The `expressions` library provides a comprehensive implementation of all expressions available in Java. This includes a wide range of operators, from simple arithmetic operations (e.g., addition, subtraction, multiplication) to more complex constructs such as ternary operators, bitwise shifts, and unary operations. By leveraging this library, the DSL gains support for a powerful and expressive syntax, allowing users to define dynamic and complex logic within their programs. However, for `guide` only the operations considered necessary such as those for booleans, numbers, and strings have been enabled. This selective activation ensures that the DSL remains lightweight and focused while still offering sufficient flexibility.

The `types` library provides an implementation of Java's type system, enabling the DSL to define and use types such as numbers, strings, and booleans. This module is a dependency of the `expressions` library, as expressions often rely on type information to perform operations and validations. By integrating `types`, the DSL ensures type safety and consistency across all operations. Similar to the `expressions` library, only the specific types required for `guide` have been enabled to reduce complexity and maintain focus on the intended use cases.

One of the key advantages of `Neverlang` is its inherent modularity. This allowed the selective customization of these libraries to suit the specific requirements of `guide`. By activating only the necessary types and operations, the `neverlang-commons` module provides a lightweight yet powerful foundation for the DSL, ensuring that it remains efficient and well-aligned with its design goals.

5.6 SPL Implementation

The Software Product Line (SPL) approach plays a central role in enabling the modularity and configurability of the system. By structuring the project into distinct, reusable

modules, the SPL allows for the generation of customized variants of DSL, ensuring that only the necessary features are included in a given configuration. This minimizes redundancy, optimizes performance, and simplifies maintenance by reducing unnecessary dependencies.

The SPL is responsible for managing feature selection and composition, ensuring that the final system instance includes only the components and layouts required for a particular use case. This is achieved through a combination of feature-based composition mechanisms and automated build configurations that dynamically assemble the selected modules into a cohesive system.

Additionally, the SPL framework includes utility programs designed to facilitate its extension. These utilities assist in defining new features, managing dependencies, and ensuring consistency between the DSL and the library. Through this structured approach, the SPL enhances the adaptability and maintainability of the architecture, making it easier to introduce new GUI elements, target different platforms, or refine existing functionality without requiring extensive modifications to the core system.

5.6.1 Feature-Based Composition

The SPL framework follows a feature-based composition approach to dynamically generate a customized version of the system based on user-defined requirements. At the core of this process is the program responsible for creating the Gradle-based project `guide-jar`. This program allows users to specify the features they want to include, ensuring that the generated system contains only the necessary components without unnecessary overhead.

Features in the SPL are divided into three main categories:

- **Output Languages**: These define the programming languages into which the GUI code will be generated.
- **Components**: These represent the various UI elements available in the generated system.
- **Layouts**: These define the structural arrangements of components within the user interface.

To ensure that the generated project functions correctly, at least one feature from each category must be selected. Additionally, the `root` feature is always required, as it serves as the entry point for GUI generation.

Once the user has selected the desired features, the system proceeds with the generation of the `guide-jar` project. This involves several steps:

1. The existing `guide-jar` directory is deleted and recreated to ensure that the build starts from a clean state.
2. A new Gradle build file is generated, including dependencies only for the selected features, keeping the project minimal and efficient.
3. A factory is created for each requested output language, ensuring that the DSL-generated UI definitions are correctly translated into the target language.

4. A CLI-based main program is generated, allowing users to interact with the system through the command line.

The CLI tool serves as the interface for compiling and executing GUI definitions written in the DSL. When running the tool, users must provide:

- The directory containing the `.guide` files.
- The name of the root module, formatted as `moduleName.root`.
- The target output language for code generation.
- The name of the output file to be generated.

To facilitate deployment and distribution, the Gradle build also includes a dependency on `shadowJar`, which allows the entire project to be packaged as a single executable JAR file. This means that the resulting system can be easily shared and used as a standalone CLI tool, functioning as a dedicated compiler for the GUIDE DSL. Thanks to this modular approach, developers can generate a custom versions of the system that match their specific needs while maintaining flexibility for future adaptations.

5.6.2 Utility Programs for Extension

To facilitate the extensibility of the framework, the SPL provides two utility programs designed to streamline the process of adding new components, layouts, or output languages. These programs automate the creation of the necessary project structure, ensuring consistency and reducing the effort required for manual setup.

Component and Layout Template Generator

The first utility program is responsible for generating the template for a new component or layout. It requires two inputs from the user:

- The name of the new component or layout.
- A specification of whether the entity being created is a component or a layout.

Based on these inputs, the program automatically generates the required module structure within the framework. Specifically, it creates:

- A new Gradle submodule under `guide-lib/components` or `guide-lib/layouts`, depending on the type of entity being added. This module serves as the abstract definition of the new component or layout.
- A corresponding submodule for the concrete implementation of the new component/layout for each supported output language.
- A Neverlang module within `guide-dsl` that defines the DSL rules required to integrate the new component/layout into the GUIDE language.

All generated modules include pre-filled template files to provide a structured starting point for implementation. Although these files contain only partial definitions and require further development, they ensure that the necessary files and dependencies are in place, reducing setup time and minimizing errors.

Output Language Template Generator

The second utility program is designed to facilitate the addition of a new output language to the framework. When executed, it creates the necessary directory structure within the `adapter` folder inside the library. The generated structure includes:

- A `core` directory containing the foundational components required to support the new language.
- A `components` directory with submodules for all existing UI components, ensuring they can be implemented in the new language.
- A `layouts` directory containing the necessary modules for layouts, following the same modular structure as components.

Similar to the component/layout generator, this utility program does not create empty directories but instead provides template files as a foundation for implementation. These files help standardize the integration of new languages while leaving room for customization. However, since they contain only placeholders, the generated modules will not compile until the implementation is completed.

By providing these utility programs, the framework ensures that extending GUIDE with new features is a straightforward and consistent process. Developers can quickly set up the required project structure and focus on implementing the actual components, layouts, or language adapters, streamlining the overall development workflow.

5.7 Example Usage of the DSL

This example expands upon the structure illustrated in Figure 4.2, translating it into a concrete implementation using the GUIDE DSL. The corresponding GUIDE DSL code is shown in Listing 5.6, which defines the UI components and their hierarchical relationships in a structured manner. Furthermore, Figures 5.4, 5.2, and 5.3 showcase the generated interfaces in three different output languages: Python, HTML, and Elixir, demonstrating the portability of the approach.

5.7.1 Description of the Example

The salary management system consists of the following UI elements:

- A **title** that serves as the heading of the application.
- A **table** displaying the existing salary records.
- A **form** for adding new salary entries, containing three input fields for:
  - First name
  - Last name
  - Salary amount
- A **text message** accompanied by a **button** that allows users to delete all salary records.

```
Module salaries {
  title = "Salaries"
  deletePrimaryColor = #color("red")
  deleteSecondaryColor = #color("#000000")

  gridLayout = GridLayout {
    Text {
      textColor: #color("blue"),
      backgroundColor: #color("#FFFF00"),
      content: title
    };
    Table {
      id: table,
      headers: ["Name", "Surname", "Salary"],
        rows: [
            "John", "Doe", "1000";
            "Jane", "Doe", "2000";
        ]
    };

    Form {
      id: addForm,
      submitButtonText: "Add salary",
      onSubmit: @addFormOnSubmit,
      fields: [
        "Name": InputText {
          id: name,
          placeholder: "Name"
        },
        "Surname" : InputText {
          id: surname,
          placeholder: "Surname"
        },
        "Salary" : InputText {
          id: salary,
          placeholder: "Salary"
        }
      ]
    };

    Text {
      textColor: deletePrimaryColor,
      backgroundColor: deleteSecondaryColor,
      content: "Delete all salaries: "
    },
    Button {
      id: deleteButton,
      textColor: deletePrimaryColor,
      backgroundColor: deleteSecondaryColor,
      content: "Delete",
      onClick: @deleteButtonOnClick
    };
  }

  root = Root {
    module: "salaries",
    title: title,
    height: 600,
    width: 800,
    callbacksFilename: "salaries_callbacks",
    layout: gridLayout
  }
}
```

Listing 5.6. *Example of a salary management system described using the GUIDE DSL. The system includes a title, a table displaying salary records, a form for adding new entries, and a text message with a button to delete all records .*

5.7.2 Implementation and Evaluation

The chosen example is a salary management system and was deliberately selected due to its balance between complexity and representativeness. It exercises all the core features of GUIDE's DSL, including hierarchical component composition, attribute management, and dynamic rendering across multiple output languages. Specifically, it incorporates diverse UI elements such as tables, forms, input fields, and buttons, arranged using a grid layout. This variety ensures that the example touches upon a wide range of functionalities, providing a meaningful benchmark for GUIDE's capabilities.

The example was considered sufficient for evaluation because it achieves broad coverage of key GUIDE functionalities. It requires:

- **Component nesting and hierarchy management**, testing the composite pattern implementation.
- **Dynamic attribute propagation**, verifying the correctness of contextual attribute handling.
- **Cross-language code generation**, ensuring that GUIDE can target multiple platforms seamlessly.
- **Callback integration**, validating event-driven interactions across supported languages.

Moreover, the salary management system poses specific challenges, such as maintaining state consistency across user interactions (e.g., adding and removing salary entries) and rendering dynamic content in the UI. Successfully implementing these aspects demonstrates that GUIDE can manage non-trivial application logic and UI updates.

The generated application functioned as expected across all supported output languages demonstrating GUIDE's flexibility and portability. The only manual effort required was the implementation of callback functions handling user interactions, such as adding or removing salaries.

This validation process confirmed that GUIDE's DSL provides a practical and expressive way to define UI structures while maintaining modularity and reusability across different platforms. The example's ability to cover multiple core functionalities, combined with its moderate complexity, makes it a convincing proof of GUIDE's robustness and applicability to more complex scenarios.

An additional observation from the evaluation process is the visual consistency across different output languages. As shown in Figures 5.2 and 5.3, the final UI maintains a nearly identical appearance despite being implemented using two distinct programming languages and styling frameworks: Bootstrap for HTML and Tailwind CSS for Elixir. This demonstrates the effectiveness of GUIDE in abstracting UI structure from language-specific details, ensuring that interfaces remain coherent regardless of the target environment. Such consistency is particularly valuable in multi-platform applications, where maintaining a uniform user experience across different technologies is essential. However, the Python (Tkinter) version, shown in Figure 5.4, exhibits noticeable differences in appearance due to the nature of desktop UI frameworks. Despite these differences, the core structure of the UI remains consistent across implementations.

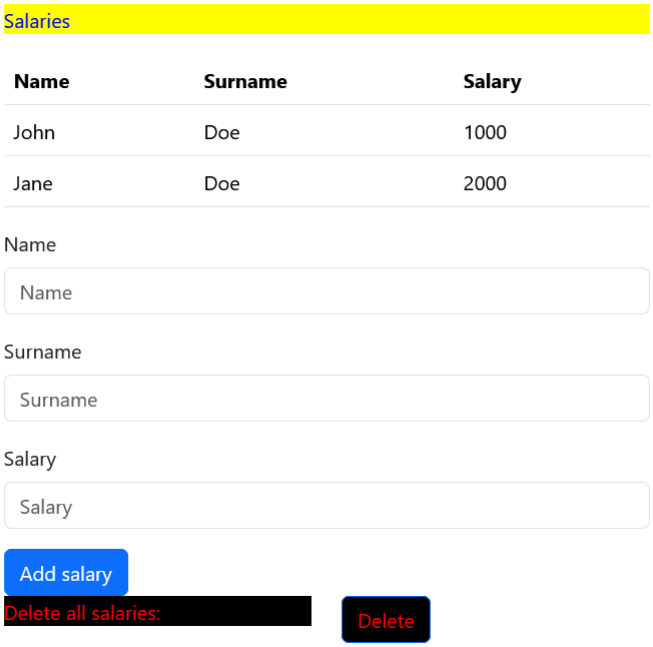

Figure 5.2. *Generated salary management system in HTML. The UI includes a title, a table displaying salary records, a form for adding new entries, and a text message with a button to delete all records.*

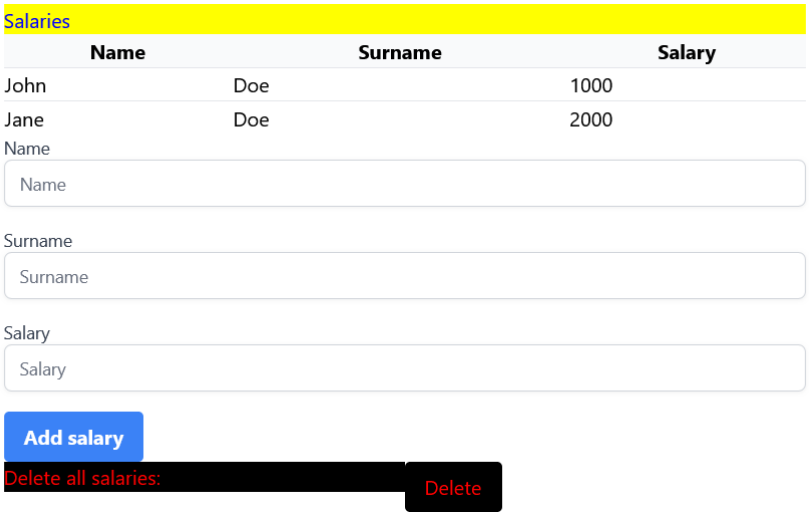

Figure 5.3. *Generated salary management system in Elixir (Phoenix LiveView). The UI includes a title, a table displaying salary records, a form for adding new entries, and a text message with a button to delete all records.*

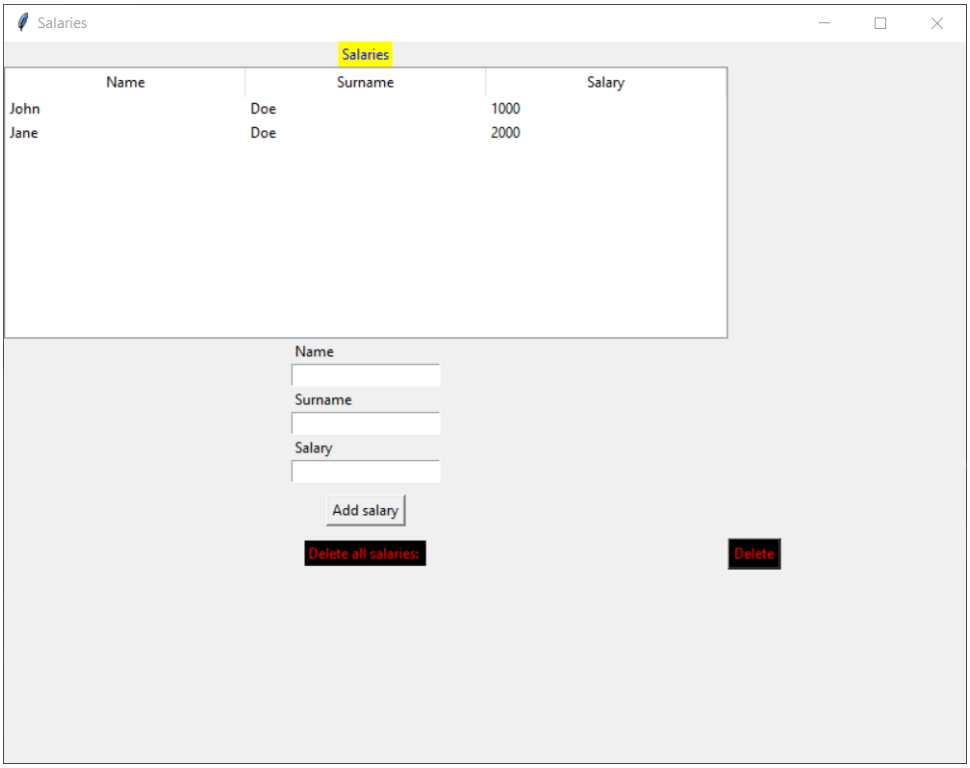

Figure 5.4. *Generated salary management system in Python (Tkinter). The UI includes a title, a table displaying salary records, a form for adding new entries, and a text message with a button to delete all records.*

6 Experiments

The purpose of this evaluation is to quantify the benefits of GUIDE in terms of modularity, reusability, and reduction of manual coding effort. By leveraging a Domain-Specific Language (DSL) and Software Product Line (SPL) techniques, GUIDE aims to streamline the development of graphical user interfaces while maintaining flexibility in feature selection.

The degree to which this goal was reached is determined on the following aspects:

First, it examines the number of possible configurations that can be generated through SPL, highlighting the modularity of the system.

Then, the evaluation measures the reduction in development effort by comparing the size of DSL specifications to the generated GUI code. By analyzing the expansion factor in terms of lines of code and characters, this study quantifies the impact of GUIDE in minimizing repetitive coding tasks and accelerating development.

Additionally, it evaluates the effort required to extend GUIDE by adding new components or target languages, comparing manual implementation with the automated approach provided by SPL.

Finally, the study assesses the effect of feature selection on storage and distribution efficiency, demonstrating that compression and dependency management play a more significant role than selective feature activation in reducing the software footprint.

These analyses provide insight into GUIDE's strengths and limitations, offering a comprehensive view of its advantages over traditional, manually written GUI code.

6.1 Modularity and Scalability

A key advantage of GUIDE's architecture is its modularity, which allows users to generate customized versions of the system by selectively enabling only the necessary features. The modular structure is based on three main feature categories:

- **Output Languages**: 3 available options (Python, HTML, Elixir).
- **Components**: 6 available components, including a mandatory `root`.
- **Layouts**: 1 available layout.

Unlike traditional monolithic approaches, GUIDE allows multiple languages and multiple layouts to be enabled simultaneously within the same configuration. This significantly increases the number of possible system variations. The total number of valid configurations can be computed using the following formula:

$$\text{Total Configurations} = (2^{\#\text{ Languages}} - 1) \times (2^{\#\text{ Layouts}} - 1) \times (2^{(\#\text{ Components}-1)} - 1)$$

Each term in the formula represents a different aspect of configurability:

- $(2^{\#\text{ Languages}} - 1)$: Since multiple languages can be selected, each of the three available output languages can either be included or excluded. This results in $2^3 = 8$ total combinations, but we subtract 1 to remove the invalid case where no language is selected, leaving us with 7 valid choices.
- $(2^{\#\text{ Layouts}} - 1)$: Since multiple layouts can also be included, this term accounts for the possible selections of available layouts. In GUIDE, there is currently only one layout, meaning this term evaluates to $2^1 - 1 = 1$, ensuring at least one layout is always selected.
- $(2^{(\#\text{ Components}-1)} - 1)$: GUIDE has six components, but the `root` component is mandatory. This leaves five optional components, each of which can either be included or excluded. This results in $2^5 = 32$ combinations, but we subtract 1 to exclude the case where no additional components are selected, yielding 31 valid combinations.

Substituting the actual values for GUIDE:

$$\text{Total Configurations} = (2^3 - 1) \times (2^1 - 1) \times (2^5 - 1) = 7 \times 1 \times 31 = 217$$

This means that GUIDE can generate up to 217 distinct system variants, each tailored to specific requirements. While some of these configurations might not be particularly useful in real-world scenarios, especially when only a limited number of components are included, they demonstrate the applicability of this architecture to more complex cases. In such contexts, where extensive component libraries and multiple target languages may introduce unnecessary complexity and increase distribution size, the SPL approach proves valuable by enabling the generation of streamlined, purpose-specific system variants. The exponential relationship between the number of selectable features and the total configurations further highlights the scalability of this approach, as adding even a few new components or layouts would significantly expand the range of potential system variations.

6.1.1 Commit Analysis for Modular Development

To evaluate the modularity of GUIDE, a commit analysis was conducted to measure the extent of modifications required when introducing new features. The goal was to determine whether GUIDE successfully isolates changes within dedicated modules, ensuring minimal impact on unrelated parts of the system. Two experiments were performed: the first involved adding a new layout, while the second focused on integrating a new output language. The results provide insight into the scalability and maintainability of the system.

Adding a New Component or Layout

The first experiment involved the automatic generation of an `absolute-layout` module using GUIDE's SPL utility. After running the process, the results of the `git diff --stat` command showed a total of 13 changed files, with 69 lines of code added and only one existing file modified as shown in Listing 6.1.

The breakdown of the modifications is as follows:

- **12 newly added files:**
  - The Neverlang syntax module for DSL integration.
  - Concrete implementations for all three output languages (Python, HTML, and Elixir).
  - The abstract definition of `absolute-layout` within the library.
  - The corresponding test module for unit testing.
  - Build scripts and factory classes required for integration.
- **1 modified file:** The only existing file altered was `Layout.java` in the SPL module, which maintains the list of available layouts. This modification ensures that the newly introduced layout is automatically recognized by future configurations generated by GUIDE.

```
guide-dsl/layouts/absolute-layout/build.gradle                                    |  3
guide-dsl/layouts/absolute-layout/src/.../AbsoluteLayoutSlice.nl                  |  1
guide-lib/adapters/elixir/layouts/absolute-layout/build.gradle                    |  3
guide-lib/adapters/elixir/layouts/absolute-layout/src/.../ElixirAbsoluteLayout.java | 10
guide-lib/adapters/html/layouts/absolute-layout/build.gradle                      |  3
guide-lib/adapters/html/layouts/absolute-layout/src/.../HtmlAbsoluteLayout.java   | 10
guide-lib/adapters/python/layouts/absolute-layout/build.gradle                    |  3
guide-lib/adapters/python/layouts/absolute-layout/src/.../PythonAbsoluteLayout.java | 10
guide-lib/layouts/absolute-layout/build.gradle                                    |  0
guide-lib/layouts/absolute-layout/src/main/.../AbsoluteLayout.java                |  5
guide-lib/layouts/absolute-layout/src/main/.../AbsoluteLayoutFactory.java         |  5
guide-lib/layouts/absolute-layout/src/test/.../AbsoluteLayoutTest.java            | 15
guide-spl/src/main/java/it/unimi/di/adaptlab/guide/spl/Layout.java                |  1
13 files changed, 69 insertions(+)
```

Listing 6.1. *`git diff --stat` output for adding a new layout to GUIDE .*

The experiment demonstrated that GUIDE successfully isolates feature additions. All necessary structural elements were created in dedicated locations without requiring manual changes to unrelated parts of the system. Some additional supporting directories, such as pre-organized resource folders, were not tracked by Git because they contained no files at this stage.

It is important to note that if the same experiment had been conducted with the addition of a new component instead of a layout, the results would have been identical. The modular structure of GUIDE ensures that new components follow the same automatic generation process, creating dedicated modules without impacting existing

ones.

While the only modification to an existing file was within the SPL module, this small level of coupling could be further minimized by making the discovery of layouts, components, and languages fully dynamic. This enhancement would eliminate the need to update the registry manually, further improving system modularity.

Adding a New Output Language

The second experiment involved adding support for a new output language, Java. This process was also performed using the SPL utility, and the results of the `git diff --stat` command showed a total of 19 changed files, with 109 lines of code added and one existing file modified as shown in Listing 6.2.

The modifications were distributed as follows:

- **18 newly added files:**
  - A new `java` directory under the `adapters` folder in GUIDE's library.
  - Concrete implementations of all existing components (`button`, `form`, `input-text`, `root`, `table`, `text`).
  - A `core` module containing essential logic for Java-based UI generation.
  - A concrete implementation of `grid-layout` for Java.
  - Corresponding build scripts to manage dependencies for each module.
- **1 modified file:** The existing file `Language.java` in the SPL module was updated to register Java as a supported language. This ensures that any future configurations including Java will automatically generate the necessary module structure.

Similar to the layout addition, the system maintained a high degree of modularity, with all changes confined to new files within the designated adapter structure. The only modification to an existing file was the update to the SPL language registry, which could also be eliminated by implementing a dynamic discovery mechanism.

Additionally, as observed in the previous experiment, some directories that would normally contain pre-organized resources for the new language were not included in the `git diff --stat` output due to Git's behavior of not tracking empty folders. These directories are part of the generated structure and would normally be populated during the implementation phase, further streamlining the development process.

Conclusion

Both experiments confirm that GUIDE adheres to strong modular development principles, as the addition of new components, layouts, and languages remains isolated within dedicated modules. The minimal impact on pre-existing files demonstrates the effectiveness of the SPL approach in managing system variability.

Future improvements could focus on further reducing manual interventions in the SPL registry files by implementing a dynamic feature discovery mechanism, allowing

```
guide-lib/adapters/java/build.gradle                                    |  0
guide-lib/adapters/java/components/build.gradle                         |  7
guide-lib/adapters/java/components/button/build.gradle                  |  3
guide-lib/adapters/java/components/button/src/.../JavaButton.java       | 10
guide-lib/adapters/java/components/form/build.gradle                    |  3
guide-lib/adapters/java/components/form/src/.../JavaForm.java           | 10
guide-lib/adapters/java/components/input-text/build.gradle              |  3
guide-lib/adapters/java/components/input-text/src/.../JavaInputText.java | 10
guide-lib/adapters/java/components/root/build.gradle                    |  3
guide-lib/adapters/java/components/root/src/.../JavaRoot.java           | 10
guide-lib/adapters/java/components/table/build.gradle                   |  3
guide-lib/adapters/java/components/table/src/.../JavaTable.java         | 10
guide-lib/adapters/java/components/text/build.gradle                    |  3
guide-lib/adapters/java/components/text/src/.../JavaText.java           | 10
guide-lib/adapters/java/core/build.gradle                               |  3
guide-lib/adapters/java/layouts/build.gradle                            |  7
guide-lib/adapters/java/layouts/grid-layout/build.gradle                |  3
guide-lib/adapters/java/layouts/grid-layout/src/.../JavaGridLayout.java | 10
guide-spl/src/main/java/it/unimi/di/adaptlab/guide/spl/Language.java    |  1
19 files changed, 109 insertions(+)
```

Listing 6.2. *`git diff --stat` output for adding a new language to GUIDE .*

new elements to be automatically detected and integrated without requiring explicit registration.

6.2 Code Reduction and Development Effort

One of the key advantages of using GUIDE is the significant reduction in the amount of code that developers need to manually write. To quantify this benefit, an experiment was conducted by implementing the salary management system example presented in the implementation chapter (Listing 5.6) using GUIDE's DSL and generating equivalent code in three target languages: Python (Tkinter), HTML (Bootstrap), and Elixir (Phoenix LiveView).

The Expansion Factor is a key metric for evaluating the effectiveness of GUIDE in reducing the manual effort required to develop a user interface. The formula used to calculate the Expansion Factor for both Lines of Code (LOC) and Code Size (CS) is presented below, where CS refers to the total number of characters in the code.

$$\text{Expansion Factor (LOC)} = \frac{\text{LOC Generated}}{\text{LOC DSL}} - 1$$

$$\text{Expansion Factor (CS)} = \frac{\text{CS Generated}}{\text{CS DSL}} - 1$$

This formula shows the increase (or decrease) in the number of lines or characters generated compared to the original DSL. A positive result indicates an expansion, while a negative result would imply a reduction.

The following table (Table 6.1) summarizes the number of lines of code (LOC) and characters (CS) required in each case, along with the calculated Expansion Factors:

Language	LOC DSL	LOC Gen.	Expansion Factor (LOC)	CS DSL	CS Gen.	Expansion Factor (CS)
Python (Tkinter)	63	55	-12.69%	1353	2517	+86.03%
HTML (Bootstrap)	63	62	-1.59%	1353	2067	+52.77%
Elixir (Phoenix)	63	81	+28.57%	1353	3232	+138.87%

Table 6.1. *Comparison between GUIDE DSL and generated code in different languages.*

From the results, it is evident that the generated code is significantly larger in terms of characters, even in cases where the LOC count remains similar or decreases slightly (e.g., Python and HTML). This is due to differences in syntax verbosity across languages and the necessity of additional boilerplate code in the generated output.

It is important to note that the DSL code in this experiment was deliberately written with additional variable allocations, even when those variables were used only once. This was done to demonstrate GUIDE's capability of handling variable definitions and to showcase a broader range of DSL features. In a real-world scenario, the DSL code could be written in fewer lines and with fewer characters while achieving the same result, making GUIDE even more efficient in reducing manual coding effort.

6.2.1 Estimated Time Savings

To quantify the practical benefits of GUIDE in terms of developer effort, we estimate the time required to manually write the generated code compared to using the DSL. Based on industry research, developers typically write code at an average rate of 20-30 LOC per hour for structured and production-ready code [9, 24].

For this analysis, we approximate that writing the DSL version of the Salary Management System took around one hour, based on manual observations. This is a rough estimate and may vary depending on the developer's familiarity with the DSL and the complexity of the UI being defined.

The estimated time savings can be calculated using the following formula:

$$\text{Estimated Saving} = 1 - \frac{\text{Time required for DSL}}{\text{Time required for manual coding}}$$

where:

- Time required for DSL is 60 minutes.
- Time required for manual coding is based on the previously computed time estimates for each target language.

The results are summarized in Table 6.2.

Language	LOC Generated	Time Manual (min)	Time DSL (min)	Estimated Saving (%)
Python (Tkinter)	55	110	60	$1 - \frac{60}{110} = 45\%$
HTML (Bootstrap)	62	124	60	$1 - \frac{60}{124} = 51\%$
Elixir (Phoenix)	81	162	60	$1 - \frac{60}{162} = 63\%$

Table 6.2. *Estimated time savings using GUIDE compared to manual coding.*

From these results, GUIDE reduces development time by an estimated 45-63%, depending on the target language. The greatest savings are observed in Elixir, likely due to the verbosity of Phoenix LiveView, whereas Python, having a more compact syntax, shows a lower percentage gain.

Although the one-hour estimate for writing the DSL is approximate, it is important to note that even with variations in this time, GUIDE still provides a significant reduction in manual coding effort. Additionally, for more complex interfaces, the time saved is expected to be even greater, as the advantages of automation scale with UI complexity.

6.2.2 Conclusion

The results demonstrate that GUIDE effectively reduces the amount of code that developers need to manually write, both in terms of lines and characters. By abstracting low-level details and automating the generation process, GUIDE not only decreases coding effort but also minimizes the likelihood of errors and inconsistencies in UI implementation. The estimated time savings further highlight the practical benefits of using GUIDE, making it a valuable tool for accelerating GUI development.

6.3 Effort Analysis for Feature Extension

An essential aspect of evaluating GUIDE's impact is understanding how much effort it reduces when adding new features. This section analyzes the effort required to extend GUIDE with a new component or a new output language, comparing the amount of manual coding needed with and without GUIDE. The goal is to quantify the reduction in development effort, both in terms of lines of code (LOC) and estimated time savings.

6.3.1 Adding a New Output Language

To estimate the effort required to add a new output language manually, we examined the existing adapters for Python, HTML, and Elixir. The total LOC across all three adapters is 1375, leading to an average LOC per language of:

$$\text{LOC}_{\text{adapter}} = \frac{1375}{3} \approx 458.33$$

When adding a new language using GUIDE's SPL utility, 108 LOC are generated automatically, as confirmed by the commit analysis in Listing 6.2. This means the actual manual effort required is:

$$\text{LOC}_{\text{manual}} = \text{LOC}_{\text{adapter}} - \text{LOC}_{\text{generated}} = 458 - 108 = 350$$

Using the same time estimation model discussed in the previous chapter (based on [9, 24]), and assuming a developer writes between 20 and 30 LOC per hour, we estimate the manual time required:

$$\text{Time}_{\text{manual}} = \frac{\text{LOC}_{\text{manual}}}{\text{LOC/hour}} = \frac{350}{20} \text{ to } \frac{350}{30} = 11.6 \text{ to } 17.5 \text{ hours}$$

The estimated effort reduction is:

$$\text{Effort Reduction} = 1 - \frac{\text{LOC}_{\text{manual}}}{\text{LOC}_{\text{adapter}}} = 1 - \frac{350}{458} \approx 23.6\%$$

6.3.2 Adding a New Component or Layout

A similar analysis was conducted to estimate the effort required to add a new GUI component. Across the seven existing components and layouts (`root`, `button`, `form`, `input-text`, `table`, `text`, and `grid-layout`), the total LOC (excluding adapters) is 2784, leading to an average LOC per component of:

$$\text{LOC}_{\text{component}} = \frac{2784}{7} \approx 397.71$$

The adapter portion of each component is estimated by taking the average size of an adapter (458 LOC) and dividing it by the number of components and layouts, then multiplying by three to account for the three output languages (Python, HTML, and Elixir):

$$\text{LOC}_{\text{adapter per component}} = \left(\frac{458}{7}\right) \times 3 = 65.43 \times 3 \approx 196.3$$

Thus, the total manual effort required to add a new component manually (including its adapters for all languages) is:

$$\text{LOC}_{\text{manual}} = \text{LOC}_{\text{component}} + \text{LOC}_{\text{adapter per component}} = 398 + 196 = 594$$

Using GUIDE, 68 LOC are generated automatically (Listing 6.1), reducing the manual effort to:

$$\text{LOC}_{\text{manual}} = 594 - 68 = 526$$

Using the same time estimation model as before, the manual implementation time is:

$$\text{Time}_{\text{manual}} = \frac{526}{20} \text{ to } \frac{526}{30} = 17.5 \text{ to } 26.3 \text{ hours}$$

The estimated effort reduction is:

$$\text{Effort Reduction} = 1 - \frac{\text{LOC}_{\text{manual}}}{\text{LOC}_{\text{component}} + \text{LOC}_{\text{adapter per component}}} = 1 - \frac{526}{594} \approx 11.4\%$$

6.3.3 Conclusion

The results indicate that GUIDE significantly reduces the manual effort required for adding both new output languages and new GUI components or layouts. While the effort reduction for new components (11.4%) is lower than for new languages (23.6%), this is expected since component additions require more customization. However, in both cases, the SPL utility reduces boilerplate work, allowing developers to focus on feature-specific logic rather than repetitive setup tasks.

6.4 Impact of Feature Selection on Storage and Distribution Efficiency

Optimizing the size of distributed software packages is an important aspect of improving efficiency, reducing download times, and minimizing resource consumption. This is particularly relevant in the context of green computing, where reducing storage and bandwidth usage translates to lower energy consumption across cloud infrastructures and end-user devices.

This section analyzes the impact of GUIDE's feature selection on the final JAR size, evaluating how much space could be saved by distributing an optimized version with only the required features.

6.4.1 JAR Size Analysis

The GUIDE JAR file is inherently compressed, reducing its size by approximately 38%. However, to analyze the actual weight of each component, measurements were conducted on the uncompressed JAR, which occupies a total of 56.87 MB (56 865 520 bytes).

The majority of this space is occupied by **Neverlang (76.5%)** and **other dependencies (22.5%)**, while GUIDE's own components collectively account for just **1% of the total size**. The overall distribution of the JAR is illustrated in Figure 6.1.

To better understand the breakdown of GUIDE's portion, Figure 6.2 provides a zoomed-in view, showing that the **DSL module contributes 0.7%**, followed by **Adapters (0.2%)** and the **Library (0.1%)**.

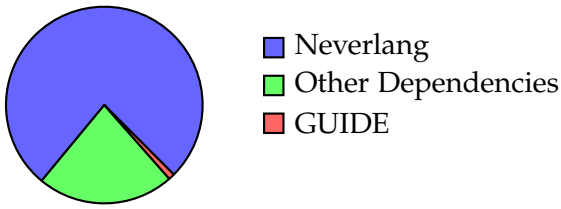

Figure 6.1. *Overall size distribution of the GUIDE uncompressed JAR.*

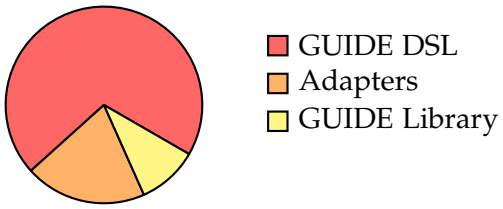

Figure 6.2. *Breakdown of GUIDE-specific components.*

6.4.2 Effect of Feature Removal

The results indicate that the majority of the JAR size is occupied by Neverlang (76.5%) and other dependencies (22.5%). The actual GUIDE components, including the DSL, library, and adapters, collectively account for only 1% of the total package size.

Since the GUIDE code contributes a minimal amount to the total JAR size, removing features does not significantly impact the overall storage footprint. Even if a version of GUIDE were generated with only a single output language, a minimal set of components, and one layout, the total size reduction would be negligible.

However, while the storage impact of feature removal is limited, this selective approach remains highly relevant in enterprise scenarios. Companies might not want to distribute parts of the software that have not been purchased or are unnecessary for specific clients. Selective feature distribution can therefore reduce the complexity of the delivered product and limit exposure to intellectual property concerns by only including relevant components.

Furthermore, the possibility of distributing Neverlang and other dependencies separately could mitigate the minimal effect of feature removal. In contexts where multiple GUIDE variants are needed, a single shared copy of Neverlang could be reused across all configurations. This would significantly reduce storage requirements and bandwidth usage, as only the JAR files corresponding to the specific features would need to be distributed for each variant.

Compression also plays a significant role in optimization. Since the JAR is already internally compressed by 38%, further reductions can be achieved through advanced distribution-level compression techniques. Nonetheless, separating core dependencies from feature-specific components represents an additional, practical strategy for optimizing storage and distribution efficiency.

6.4.3 Modularizing Dependencies for Efficient Distribution

While selective feature removal has a limited impact on overall storage due to the dominance of core dependencies, further distribution efficiency can be achieved by modularizing these dependencies. In particular, Neverlang accounts for 76.5% of the uncompressed JAR size, making it a prime candidate for separate distribution.

By distributing Neverlang and other major dependencies independently, multiple GUIDE variants could share a single instance of these components. This approach

is especially beneficial in enterprise environments where different clients require customized versions of GUIDE. Instead of distributing multiple complete JARs, each containing redundant copies of Neverlang, the core can be shared while only the specific JAR files corresponding to the desired features are distributed per variant.

Such modularization not only reduces storage requirements but also simplifies maintenance. Updates to core dependencies would not require redistributing all variants, as the shared modules could be updated separately. Moreover, this approach aligns with best practices in software distribution, where decoupling core frameworks from application-specific logic leads to more efficient deployment pipelines and lower operational costs.

In conclusion, modularizing core dependencies like Neverlang presents a scalable and efficient strategy for distribution, particularly when managing multiple GUIDE configurations across diverse environments.

6.4.4 Advanced Compression Strategies for Distribution Efficiency

While GUIDE's feature selection mechanism enables modularity, its impact on storage reduction is limited. The majority of the JAR size is occupied by dependencies such as Neverlang, making feature removal an ineffective strategy for reducing distribution size. The most effective way to optimize storage and bandwidth usage is through distribution-level compression techniques, rather than feature selection.

One possible approach is adopting advanced compression algorithms like Zstandard (zstd) [15] or Brotli [2], which provide better compression ratios and faster decompression speeds than traditional ZIP. These algorithms have been widely adopted for reducing software distribution overhead while maintaining high performance.

Another strategy is delta compression, which minimizes update sizes by transmitting only the differences between two versions of a software package. This method has been successfully applied in various contexts, such as Google's Courgette system[1,2], which achieves significant bandwidth savings for software updates. In environments where software updates are frequent, this approach reduces redundant data transfers, lowering network load and power consumption.

Reducing unnecessary dependencies is another optimization route. Dependency pruning and tree-shaking techniques, used extensively in JavaScript frameworks like Webpack[3], analyze the dependency graph to eliminate unused modules, thereby reducing the final package size. This concept could be applied to GUIDE by dynamically selecting only the required portions of Neverlang. Since GUIDE's largest storage overhead comes from its dependencies, a more selective inclusion of runtime components could enhance efficiency while reducing storage requirements on distributed systems.

Finally, executable compression techniques like UPX[4] (Ultimate Packer for Executables) offer an additional reduction in binary size while preserving functionality.

[1] https://blog.chromium.org/2009/07/smaller-is-faster-and-safer-too.html
[2] https://www.chromium.org/developers/design-documents/software-updates-courgette/
[3] https://webpack.js.org/
[4] https://upx.github.io/

Although these methods introduce slight CPU overhead during decompression, they can significantly reduce storage requirements, which is beneficial in cloud-based deployments where energy efficiency is a priority.

Since data transmission is a major factor in energy consumption for cloud services and distributed systems [4], reducing package size directly contributes to lower energy usage and improved sustainability. Large software packages require greater bandwidth, increase the load on network infrastructure, and lead to higher power consumption in data centers due to prolonged data transfer and storage demands. By minimizing the amount of data that needs to be distributed, optimized compression and dependency management not only enhance software efficiency but also reduce the carbon footprint associated with large-scale deployments.

Future optimizations in GUIDE could involve adopting a hybrid approach, combining dependency reduction with more efficient compression methods. By improving storage and bandwidth efficiency, these optimizations not only reduce distribution overhead but also contribute to sustainable software practices, making GUIDE more suitable for resource-constrained environments.

6.4.5 Conclusion

The evaluation of GUIDE's storage and distribution efficiency has revealed that feature selection has a limited impact on the overall package size, as the majority of the JAR is occupied by external dependencies, particularly Neverlang. Even with a minimal set of features, the total storage footprint remains largely unchanged, making feature removal alone an ineffective optimization strategy.

A more practical approach to improving distribution efficiency lies in modularizing dependencies. By distributing core dependencies like Neverlang separately, multiple GUIDE variants can share a single instance of these components. This approach is especially beneficial in enterprise contexts, where clients may require tailored versions of the software. Selective distribution ensures that only the relevant components are delivered, reducing redundancy and addressing intellectual property concerns.

Additionally, distribution-level compression techniques, such as advanced compression algorithms and delta updates, provide further opportunities for optimization. These methods are more effective than selective feature activation, as they target the actual storage and bandwidth demands during deployment.

Future work could explore dependency minimization strategies and alternative runtime environments to reduce the reliance on large core components like Neverlang. Such improvements would further enhance GUIDE's scalability and adaptability, making it a more efficient solution for delivering customized graphical user interface generators across diverse environments.

7 Related Work

The development of graphical user interfaces (GUIs) has been extensively studied through various approaches, often focusing on specific aspects such as code generation, modularity, and domain-specific abstractions. This chapter reviews relevant scientific works that relate to the key concepts underlying this thesis: Domain-Specific Languages (DSLs), Software Product Lines (SPLs), and graphical user interface generation.

While numerous studies explore the intersection of two of these areas, such as DSLs for GUI modeling, SPLs for managing variability in user interfaces or automated GUI generation, there is a noticeable lack of research that combines all three dimensions. This thesis aims to fill that gap by integrating DSLs, SPLs, and GUI generation into a unified framework. To contextualize this contribution, the following sections group related works based on how they combine these core aspects.

The chapter is organized as follows: first, it discusses works focusing on DSLs for GUI modeling and generation, highlighting how DSLs can abstract and simplify UI definitions. Next, it examines SPL-based approaches for GUI engineering, showcasing how SPLs handle variability and modularity in interface development. An additional section introduces studies that combine SPL and DSL, analyzing how their integration enhances flexibility and reusability in software product lines. A further section explores automated GUI generation techniques, focusing on how automation impacts development time and consistency.

7.1 DSLs for GUI Modeling and Generation

Domain-Specific Languages (DSLs) have gained attention for their ability to simplify the modeling and generation of Graphical User Interfaces (GUIs) by providing tailored abstractions for specific domains. Several approaches have explored the intersection of DSLs and GUI generation, each addressing different aspects of the design, analysis, and implementation processes.

Bacíková et al. in *Defining Domain Language of Graphical User Interfaces* [6] propose a methodology that derives DSL grammars directly from existing user interfaces. Their approach relies on the DEAL (Domain Extraction ALgorithm) method, which traverses component-based GUIs to extract domain-relevant information and generates corresponding DSL grammars. The authors emphasize that GUIs inherently define domain languages, where components like text fields, buttons, and sliders can be translated into grammar productions. However, a limitation of this approach is the

lack of built-in modularity and variability management. Although DEAL effectively generates DSLs from GUI structures, it does not provide mechanisms to customize or extend the generated DSLs for varying application requirements.

Similarly, Bacíková and Porubän in *DSL-driven Generation of Graphical User Interfaces* [7] extend the DEAL methodology by demonstrating the reverse process: generating GUIs from previously extracted DSLs. By integrating the DEAL tool with the iTask system, they show how domain knowledge encapsulated in DSLs can be leveraged to produce new applications. This approach highlights the reusability of DSLs across multiple applications, preserving domain models during migration to new platforms. However, the generated GUIs are limited in terms of flexibility, as the underlying DSLs are monolithic and do not account for variability across different user requirements. Moreover, non-standard GUI components and custom functionalities require manual adjustments, limiting the automation level.

In *A Textual Domain Specific Language for User Interface Modelling* [22], the authors introduce a textual DSL designed specifically for modeling UIs. This DSL offers a declarative syntax that captures the structure and behavior of user interfaces at a high level of abstraction. The primary advantage of this approach lies in its readability and ease of use, enabling rapid prototyping of UI designs. However, like the previous works, it lacks explicit support for modularity and variability management. As a result, adapting the DSL to accommodate new requirements or extending it to support additional UI components demands significant manual effort.

Collectively, these works demonstrate the potential of DSLs in simplifying GUI generation by abstracting away low-level implementation details. However, a common limitation across all approaches is the absence of mechanisms for managing variability and modularity-essential features when targeting diverse application requirements. This gap suggests that integrating concepts from Software Product Lines (SPLs) could address these shortcomings by enabling systematic reuse and configuration of UI components. The approach presented in this thesis builds upon these insights by combining DSLs with SPL principles, offering a modular and configurable framework for generating GUIs across multiple programming languages and platforms.

7.2 SPL Approaches in GUI Engineering

Software Product Lines (SPLs) have been widely applied to manage variability and reuse in software development. However, when applied to Graphical User Interfaces (GUIs), SPLs introduce unique challenges due to the need for balancing automation and usability. The reviewed works explore how SPL techniques have been adapted to address these issues in UI engineering, focusing on strategies for managing UI variability and ensuring user-friendly designs.

Pleuss et al. in *User Interface Engineering for Software Product Lines - The Dilemma between Automation and Usability* [25] explore the tension between fully automated UI generation and the usability of the resulting interfaces. While SPL approaches excel at automating the derivation of product variants, GUIs demand more than functional

completeness. Usability factors such as layout organization, visual coherence, and interaction patterns must also be considered. The authors emphasize that purely automated derivation can result in poorly structured interfaces when UI elements are removed based on feature deselection. This occurs because the removal of certain features can disrupt the overall visual and interactive balance of the interface, leading to degraded user experiences. To address these issues, they propose a layered abstraction approach involving Task Models, Abstract UI Models (AUI), and Concrete UI Models (CUI). These abstraction layers allow developers to refine usability aspects while still leveraging automation at the lower levels of UI generation. However, despite these refinements, maintaining consistency across multiple abstraction levels remains a complex task, especially for large and highly configurable product lines.

In a complementary study, Abdul Malik et al. in *Proposed User Interface Generation for Software Product Lines Engineering* [27] focus on automating UI generation using the Interaction Flow Modeling Language (IFML). IFML enables the modeling of user interactions separately from the application's functional logic, making it easier to represent UI variability. By leveraging IFML, their approach allows the generation of adaptive UIs that adjust automatically based on selected product features. The authors highlight that their methodology supports dynamic customization of interface layouts and navigation flows while maintaining a high level of automation. One notable strength of this approach is its ability to handle UI reconfiguration without extensive manual intervention, thereby reducing development time and effort. However, the reliance on IFML introduces certain limitations, particularly when dealing with non-standard UI components or when deep customization beyond the IFML specification is required. These cases may still necessitate manual adjustments, limiting the scalability of the approach for highly specialized applications.

Both works underscore a common challenge in SPL-based UI engineering: balancing the benefits of automation with the need for customized, usable interfaces. While abstraction techniques such as those proposed by Pleuss et al. [25] provide mechanisms for addressing usability concerns, they add complexity to the development process. Conversely, the IFML-based approach of Abdul Malik et al. [27] offers a more streamlined solution but is constrained by the expressiveness of the modeling language. Together, these studies illustrate that achieving an optimal balance between automation and usability remains a key area for further research in the field of SPL-driven UI generation.

7.3 SPL and DSL: Synergy for Software Engineering

The combination of Software Product Lines (SPLs) and Domain-Specific Languages (DSLs) offers a promising approach to enhance software reuse, modularity, and configurability. While DSLs provide high-level abstractions tailored to specific domains, SPLs offer systematic mechanisms for managing variability and producing customized software products efficiently. The reviewed works in this section explore how these two paradigms can be integrated to improve language reuse and automate DSL construction.

White et al., in *Improving Domain-specific Language Reuse with Software Product-line Techniques* [31], propose a methodology to enhance DSL reuse by applying SPL techniques. The paper highlights the challenges associated with traditional DSL development, where monolithic and tightly coupled language implementations hinder adaptability and reuse. To address this, the authors introduce a feature-oriented approach where DSLs are decomposed into modular language components, each representing a specific language feature. These features are managed within a product line, enabling the automatic generation of DSL variants based on selected feature configurations. This approach significantly reduces development effort by promoting reuse of language features across different DSLs and domains. The paper also discusses how the use of feature models helps manage dependencies between language components, ensuring consistency during the generation of customized DSL instances. However, the authors acknowledge that increasing the granularity of language features can lead to complex dependency management, posing challenges in maintaining coherence among highly modular components.

Complementing this work, Huang et al., in *Automated DSL Construction Based on Software Product Lines* [21], present a framework that automates DSL construction using SPL principles. The proposed framework captures language variability through feature models and employs automated processes to generate both the syntax and semantics of DSLs tailored to specific requirements. This automation reduces the manual effort required in DSL development and ensures consistency across generated language instances. One of the key contributions of this paper is the demonstration of how the automated generation process can handle both syntactic and semantic aspects of DSLs, providing fully functional language implementations with minimal human intervention. The framework's modular nature also facilitates the extension of DSLs with new features, enhancing adaptability to evolving domain requirements. Nevertheless, the authors note that while the framework effectively automates the generation of standard language features, complex semantic behaviors still require manual refinement, limiting the scope of full automation in certain scenarios.

Both works underscore the potential of integrating SPL concepts into DSL development. White et al. emphasize the role of feature-based decomposition in promoting language reuse, while Huang et al. focus on automating the construction of DSLs through feature-driven generation. Together, these approaches highlight the benefits of modularity, reuse, and automation in DSL development, although challenges related to managing feature dependencies and supporting complex semantics remain areas for future exploration.

7.4 Automated GUI Generation Techniques

Automated generation of Graphical User Interfaces (GUIs) has become a critical area of research, aiming to streamline development processes, ensure consistency, and reduce manual coding efforts. Two notable approaches to automate GUI generation are the use of Domain-Specific Languages (DSLs) specifically designed for GUI modeling and the

adoption of XML-compliant user interface description languages. The following works explore these approaches, highlighting their advantages, limitations, and potential applications.

Huang et al., in *Automatized Generating of GUIs for Domain-Specific Languages* [5], propose a framework that employs DSLs to automate the generation of GUIs. The approach focuses on reducing the complexity of GUI development by allowing developers to describe user interfaces using a high-level, domain-aligned syntax. The framework takes DSL specifications and automatically generates corresponding GUI code, eliminating the need for repetitive manual coding. The authors demonstrate how this methodology enhances productivity and ensures consistency across applications, as developers can focus on high-level design rather than low-level implementation details. One of the key contributions of the paper is the detailed explanation of how the DSL integrates with underlying GUI frameworks, supporting automatic code generation across multiple platforms. However, the study also identifies certain challenges. For example, ensuring that the DSL can express complex GUI behaviors without becoming overly complicated is a non-trivial task. Additionally, adapting the framework to new GUI frameworks or languages requires additional development effort, potentially limiting its extensibility.

A complementary perspective is provided by the comprehensive survey *A Review of XML-compliant User Interface Description Languages* [28], which examines various XML-based languages designed for GUI specification. The paper reviews multiple XML-compliant languages, discussing their expressiveness, platform dependency, and suitability for different application scenarios. XML-based languages provide a standardized way to represent user interfaces, separating the UI definition from application logic and thereby enhancing modularity and maintainability. The review identifies key advantages of XML-based approaches, such as their widespread adoption, ease of integration with various development tools, and ability to represent static user interface components effectively. However, the authors also point out notable limitations. XML descriptions tend to become verbose when modeling dynamic behaviors or complex interactions, potentially increasing the development burden rather than reducing it. Furthermore, while XML-based languages excel in defining static layouts, they often lack the flexibility required for representing dynamic, domain-specific behaviors without extensive customization.

These two approaches DSL-driven GUI generation and XML-compliant UI description highlight different trade-offs. DSLs provide higher abstraction levels tailored to specific domains, enabling more flexible and domain-aligned GUI development. However, they require careful language design to balance expressiveness and simplicity. On the other hand, XML-based languages offer standardization and ease of integration but may lack the flexibility and conciseness needed for more complex applications. The reviewed works suggest that combining these two methodologies could potentially leverage the strengths of both, providing a robust solution for automated GUI generation that balances flexibility, standardization, and ease of use.

8 Conclusions

The development of GUIDE has demonstrated the effectiveness of combining Domain-Specific Languages (DSLs) and Software Product Lines (SPLs) to streamline the generation of graphical user interfaces (GUIs) across multiple programming languages. By providing a high-level abstraction for UI definition, GUIDE allows developers to focus on the structure and behavior of interfaces rather than the intricacies of target languages. The modular design, enabled by SPL principles, ensures flexibility in selecting only the necessary components, reducing complexity and improving maintainability. The evaluation of GUIDE confirms its ability to simplify the development process, reducing the amount of manually written code and improving code reusability across different platforms.

One of the key strengths of GUIDE is its ability to reduce development effort by automating the generation of UI code. The evaluation results indicate that GUIDE significantly decreases the amount of manually written code, both in terms of lines and characters, across different target languages. Additionally, its modular architecture, driven by SPL principles, allows developers to include only the necessary features, improving maintainability and scalability. The ability to generate equivalent UI structures across different platforms, as demonstrated in the implementation examples, highlights GUIDE's flexibility in adapting to multiple technologies while maintaining a consistent design approach.

Despite its advantages, GUIDE presents some limitations. The modular approach of SPL allows for selective feature inclusion, but the actual storage footprint reduction remains minimal due to the significant size of external dependencies, particularly Neverlang. This limits the effectiveness of feature selection as a means of optimizing distribution efficiency. However, while feature removal has little impact on storage, the ability to distribute only the necessary components remains an important advantage. In enterprise scenarios, companies may prefer to provide customized software packages containing only the purchased or required features, reducing the complexity of the delivered product. This selective approach not only improves usability by eliminating unnecessary functionality but also mitigates intellectual property concerns by distributing only relevant components.

Future work on GUIDE could focus on several key improvements. First, generating callback function templates with empty declarations could reduce setup time, allowing developers to focus only on implementing business logic. Second, optimizing the dependency management system such as distributing Neverlang separately could enhance storage and deployment efficiency, making GUIDE more suitable for large-scale

applications. Additionally, extending GUIDE to support more GUI frameworks and programming languages would improve its applicability across different development ecosystems. Another potential enhancement is the development of a visual UI editor, allowing users to create interfaces interactively without writing DSL code manually. This feature would make GUIDE more accessible to non-programmers and streamline the UI creation process. Finally, enhancing the DSL with more advanced layout options and UI behaviors would increase its expressiveness, enabling the creation of more complex user interfaces while maintaining the simplicity of the current approach.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[2] Jyrki Alakuijala and Zoltan Szabadka. Brotli Compressed Data Format. RFC 7932, July 2016.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, April 2013.

[4] Jayant Baliga, Robert W. A. Ayre, Kerry Hinton, and Rodney S. Tucker. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 99(1):149–167, 2011.

[5] Michaela Bačíková, Dominik Lakatos, and M. Nośăl. Automatized generating of guis for domain-specific languages, 01 2012.

[6] Michaela Bačíková, J. Porubän, and Dominik Lakatos. Defining domain language of graphical user interfaces. *OpenAccess Series in Informatics*, 29:187–202, 01 2013.

[7] Michaela Bačíková and Jaroslav Porubän. Dsl-driven generation of graphical user interfaces. *Central European Journal of Computer Science*, 4(4):204–221, 2014.

[8] Tim Berners-Lee. Information technology — document description and processing languages — hypertext markup language (html), 2000. ISO/IEC 15445:2000.

[9] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall advances in computing science and technology series. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[10] Tim Bray, François Yergeau, Michael Sperberg-McQueen, Jean Paoli, and Eve Maler. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. https://www.w3.org/TR/2008/REC-xml-20081126/.

[11] Walter Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Proceedings of the 11th International Conference on Software Composition (SC'12)*, Lecture Notes in Computer Science 7306, pages 162–177, Prague, Czech Republic, 31st of May-1st of June 2012. Springer.

[12] Walter Cazzola and Davide Poletti. DSL Evolution through Composition. In *Proceedings of the 7th ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'10)*, Maribor, Slovenia, on 23rd of June 2010. ACM.

[13] Walter Cazzola and Edoardo Vacchi. Neverlang 2: Componentised Language Development for the JVM. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Proceedings of the 12th International Conference on Software Composition (SC'13)*, Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.

[14] Donald Chamberlin. Information technology — database languages sql — part 1: Framework (sql/framework), 2023. ISO/IEC 9075-1:2023.

[15] Yann Collet and Murray Kucherawy. Zstandard Compression and the 'application/zstd' Media Type. RFC 8878, February 2021.

[16] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Martin Fowler's Blog, May 2005.

[17] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. Addison Wesley, September 2010.

[18] Nadine Fröhlich and Georg Paul. Lightweight Generation of User Interfaces. In *Proceedings of the 2nd International Scientific Conference on Computer Science*, Chalkidiki, Greece, September 2005.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Oscar M. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, Lecture Notes in Computer Science 707, pages 406–431, Kaiserslautern, Germany, July 1993. Springer.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.

[21] Changyun Huang, Ataru Osaka, Yasutaka Kamei, and Naoyasu Ubayashi. Automated dsl construction based on software product lines. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 1–8, 2015.

[22] Mart Karu. A textual domain specific language for user interface modelling. In Tarek Sobh and Khaled Elleithy, editors, *Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering*, pages 985–996, New York, NY, 2013. Springer New York.

[23] Linda M. Northrop, Paul C. Clements, Felix Bachmann, John K. Bergey, Gary Chastek, Sholom G. Cohen, Patrick Donohoe, Lawrence G. Jones, Robert W. Krut, Jr., Reed Little, John McGregor, and Liam O'Brien. *A Framework for Software Product Line Practice, Version 5.0*. Software Engineering Institute, 2012.

[24] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2004.

[25] Andreas Pleuss, Benedikt Hauptmann, Deepak Dhungana, and Goetz Botterweck. User interface engineering for software product lines: the dilemma between automation and usability. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 25–34, New York, NY, USA, 2012. Association for Computing Machinery.

[26] Florian Rivoal, Tab Atkins Jr., Elika Etemad, Chris Lilley, and Sebastian Zartner. CSS snapshot 2024. Technical report, W3C, February 2025. https://www.w3.org/TR/2025/NOTE-css-2024-20250225/.

[27] Siti Ina Sakinah, Hafiyyan Sayyid Fadhlillah, Ade Azurat, and Maya R. A. Setyautami. Proposed user interface generation for software product lines engineering. In *2018 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, pages 481–486, 2018.

[28] Nathalie Souchon and Jean Vanderdonckt. A review of xml-compliant user interface description languages. In Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors, *Interactive Systems. Design, Specification, and Verification*, pages 377–391, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[29] Edoardo Vacchi and Walter Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures*, 43(3):1–40, October 2015.

[30] Edoardo Vacchi, Diego Mathias Olivares, Albert Shaqiri, and Walter Cazzola. Neverlang 2: A Framework for Modular Language Implementation. In *Proceedings of the 13th International Conference on Modularity (Modularity'14)*, pages 23–26, Lugano, Switzerland, 22nd-25th of April 2014. ACM.

[31] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt. Improving domain-specific language reuse with software product line techniques. *IEEE Software*, 26(4):47–53, 2009.