MODELING VARIABILITY AND COMPOSITIONALITY OF WEB USER INTERFACES AT A DOMAIN ENGINEERING LEVEL

JONATHAN DEYRSON ORREGO LÓPEZ

Software, Master of Engineering (M.Eng.)

Thesis

Advisor

Ph.D. RAÚL MAZO PEÑA

UNIVERSIDAD EAFIT

ESCUELA DE INGENIERÍAS

MAESTRÍA EN INGENIERÍA

MEDELLÍN

2023

CONTENTS

LIST OF FIGURES

ABSTRACT

The development of *Web User Interfaces (WUIs)* is constantly evolving by incorporating new methodologies in the context of Web and product design. Although the Web design process has demonstrated successful use cases by improving user experience, this process is primarily focused on personalization rather than modeling a WUI product at scale. A *Software Product Line Engineering (SPLE)* framework has been explored to incorporate the process of modeling *variability* and *compositionality* of WUIs. These modeling techniques have proven to be applicable in the development of WUIs although these techniques have not been applied simultaneously. With the help of *VariaMos* framework and Web application, a *WUI Modeling Language* has been designed allowing programmers to manage both variability and compositionality of WUIs at a domain engineering level. Mainly, this study will demonstrate to what extent a use case can be modeled by using the implemented WUI Modeling Language.

Keywords: user interface, web interface, variability, compositionality, software product lines, domain engineering

RESUMEN

El desarrollo de *Interfaces de Usuario Web* (WUIs, por sus siglas en inglés) está en constante evolución al incorporar nuevas metodologías en el contexto del diseño Web y de producto. Aunque el proceso de diseño Web ha demostrado casos de use exitosos al mejorar la experiencia del usuario, este proceso se centra ante todo en la personalización en lugar del modelado de un producto WUI a gran escala. Para ello, hemos explorado un marco de *Ingeniería de Líneas de Productos de Software* (SPLE, por sus siglas en inglés) con el fin de incorporar el proceso de modelado de la *variabilidad* y la *composicionalidad* de WUIs. Estas técnicas de modelado han demostrado ser aplicables en el desarrollo de WUIs; sin embargo, no se han implementado simultáneamente. Con la ayuda de del marco de trabajo *VariaMos* y su aplicación Web, hemos diseñado un *Lenguaje de Modelado de WUIs* que permite a los programadores gestionar tanto la variabilidad como la composicionalidad de WUIs a nivel de ingeniería de dominio. Principalmente, este estudio demostrará de qué manera se puede modelar un caso de uso utilizando el Lenguaje de Modelado de WUIs implementado.

Palabras clave: interfaz de usuario, interfaz web, variabilidad, composicionalidad, líneas de productos de software, ingeniería de dominio

CHAPTER 1. INTRODUCTION

MOTIVATION

Web User Interfaces (WUIs) are essential units of most software systems. From an architectural standpoint, a WUI is introduced as the client or presentation layer of a software product, allowing end users to interact with by sending, storing, or requesting information. WUIs are part of a broader set of User Interfaces (UIs), including Touch User Interfaces (TUIs),[1] Voice User Interfaces (VUIs),[2] Command Line Interfaces (CLIs), and Graphical User Interfaces (GUIs). Additionally, augmented reality and virtual reality UIs have been another emerging area of interest seeking to enhance the user experience by incorporating Artificial Intelligence (AI) (Schmidt, Mayer, & Buschek, 2021). See *Figure 1,* for a broader classification of UIs.

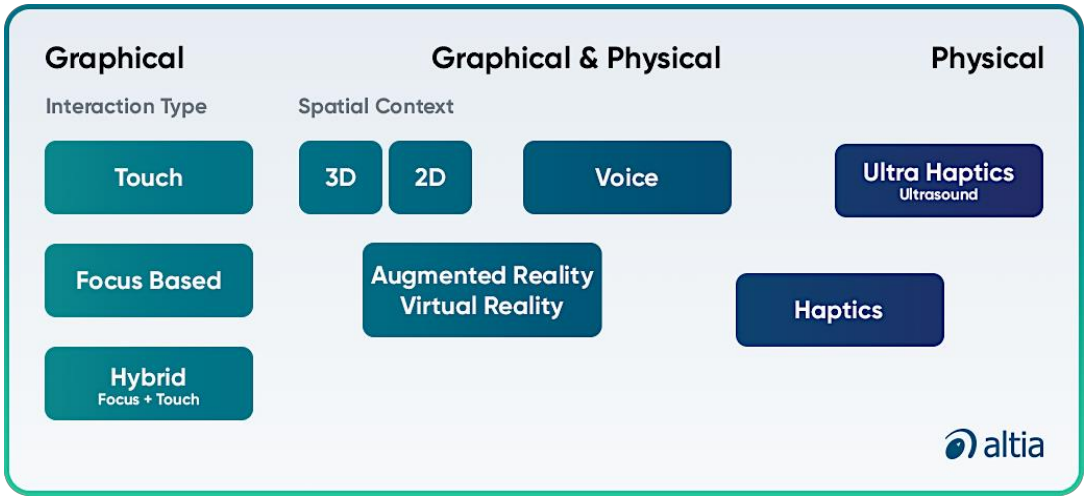

Figure 1. Types of User Interfaces[3]

[1] For further information on TUIs consider Wigdor (2010) study on Architecting Next-Generation User Interfaces.

[2] For research studies on VUIs consider Oliveira, Bezerra, Freitas, and Maciel (2015) on the Adoption of Software Product Line to a Voice User Interface Environment.

[3] Types of User Interfaces: Modern UI Design by Altia

Recently, WUIs has become the standard within the Web allowing users to consume and create highly interactive and dynamic contents. Specifically, a WUI runs on a web server enabling end users to interact with the application content throughout a Web browser. WUIs differ from WUIs in that these interfaces followed the early textual-base design where WUIs were limited to be installed and run on the computer's localhost (Hanus & Kluß, 2008). Then, early Web pages were also largely text-based, as the internet connectivity and technology were limited. There were not sophisticated designs, and the page structure was defined by the basic HTML tags for headers, paragraphs, and links. In this study, WUIs are our main concern as these interfaces exhibit two of the main modeling properties we will focus henceforth, *variability* and *compositionality* within the context of *Software Product Lines (SPLs)*.

In recent years, SPLs Engineering *(also known as SPLE)* has emerged as a well-known methodology for developing and managing software systems that can be easily adapted to different requirements or environments. In the context of SPLE, variability is defined as the ability to manage, configure, and derive product instances of a software product by customizing and extending the software functionalities and features. The aim of variability is to capture and represent the commonalities and differences among the products in a product line, enabling the management and configuration of robust software products or software product families. This study provides an introduction on variability in the context of SPLs, focusing on its fundamental challenges and capabilities for modeling a WUI along with the incorporation of compositional techniques.

Within the software engineering context, compositionality refers to the concept that a complex software system can be designed by composing functional, self-contained components as its fundamental building blocks. This notion is based on the principle that a system's behavior and structure can be formalized by understanding the system's individual components and the way they interact with each other. By breaking down a complex and robust software system into smaller components, programmers can easily reason about, modify, and extend the system and minimize unexpected breaking changes. Compositionality is a fundamental principle in modern engineering and computing, and it is playing a critical role in the design and implementation of software systems including WUIs.

Overall, the process of designing and implementing WUIs has considerably evolved, and it has changed along with the incorporation of the new Web and product design methodologies. While product design has brought substantial improvements to the Web design job, the process of modeling and engineering a WUI has been taken for granted or not fully explored. Specifically, we refer to the incorporation of variability and compositionality techniques into the process of modeling a WUI. With the help of VariaMos framework and Web toolkit, a *WUI Modeling Language* has been designed and implemented allowing programmers to model both variability and compositionality of WUIs at a domain engineering level.

CHAPTER 2. BASIC CONCEPTS

WUI DESIGN OVERVIEW

Since the early stages of the World Wide Web, WUIs have become an essential aspect of any software product, allowing users to interact with a computer or another user through highly interactive mechanisms and reach content. The WUI design process has positively improved the user experience due to the technological advancements in Human-Computer Interaction (HCI), Web development, and product design. However, with the advent of more complex and robust systems, the design and development of WUIs has become an often non-trivial and exhaustive task that involves many actors and other stages not fully explored yet.

For instance, designing and developing a WUI requires a multidisciplinary team, consisting of Web designers, Web developers, and Subject-Matter Experts (SMEs), to mention a few. During the design process, the Web designer sketches a visually appealing and user-friendly interface considering that a well-designed interface should be intuitive, easy to learn, and provide feedback to users. Consequently, both designers and developers need to comprehend the principles of product design to implement interfaces that meet not only the users' needs, but also the functional requirements.

However, despite the significant advancements in WUI design and development, we have evidenced there is an increasing need for a more engineering approach to

incorporate into this process. Currently, traditional design of user interfaces entirely relies on Web design techniques which may lack the engineering standards as this process is mainly focused on personalization and aesthetics. This lack of a standardized process and engineering rigor can lead to software malfunctioning or product delivering overdue. Thus, programmers and software engineers must take an active role in the development of user interfaces to ensure that the final product is not only visually appealing but meets the software quality standards.

Overall, the evolution of user interfaces has brought significant improvements in user experience and product design. However, the need for a more engineering approach is increasingly important as WUI becomes highly complex and robust. That is why, we have proposed a complementary modeling stage within a Software Product Line (SPL) engineering framework.

VARIABILITY IN SOFTWARE PRODUCT LINES (SPLs)

Software Product Lines (SPLs) is a modern approach to software engineering that aims to improve the software development process by increasing software reusability and reducing time-to-market. Thus, SPLs are designed to be easily customizable to meet the specific needs of different customers or market segments. One of the key concepts behind SPLs is *variability,* which refers to the ability to configure and manage software specific features which will later derive in other software products. When using SPLs, variability is usually expressed in terms of *features.*[4] A feature within a SPL has been defined as “a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems” (Kang, Cohen, Hess, Nowak, & Peterson, 1990).

To accomplish SPLs variability, industry has commonly applied a framework as the one proposed by Mazo, 2018. Although this software engineering framework has been initially conceived to outline the fundamentals of a SPL, we consider this more applicable framework to instantiate our WUI modeling process. *Figure 2* provides a summary of the SPL engineering framework.

[4] Alternatively, in this study we will mainly refer to software feature as a *component* since this concept is better aligned with the WUI componentization model.

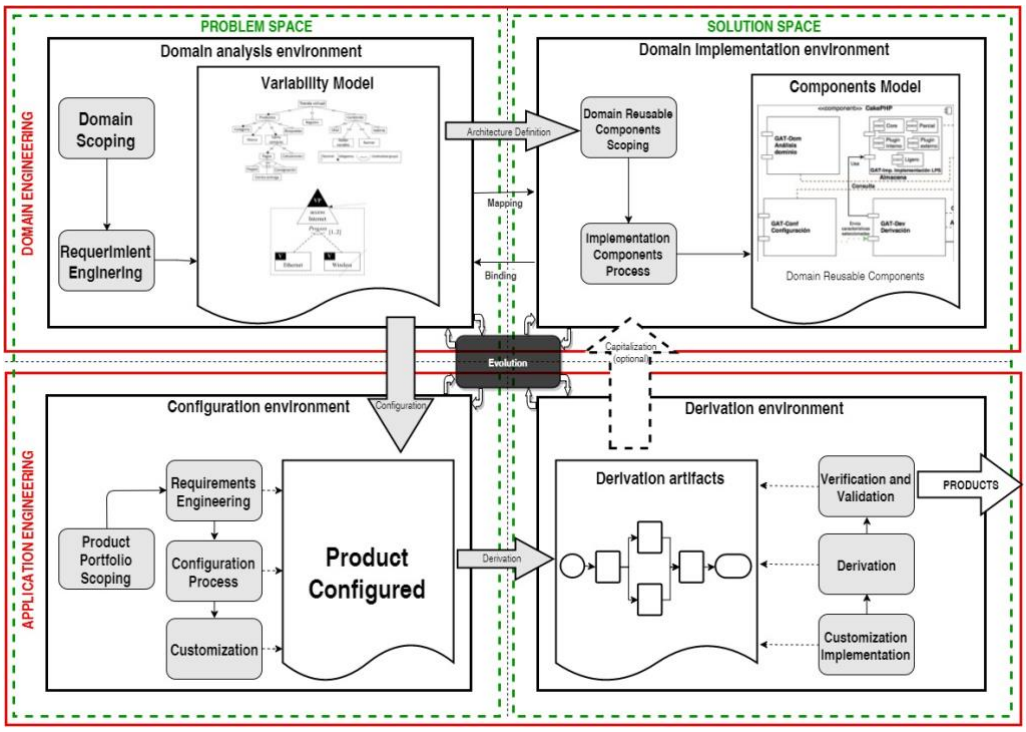

Figure 2. Software Product Line Engineering Framework

Similar to other engineering disciplines, and once the scope of the SPL has been defined, the operational activities of the SPL are initialized. This stage comprises two main perspectives: *the problem space* and *the solution space*. Nonetheless, SPL implements these two perspectives through two major processes or levels: *domain engineering* and *application engineering levels*.

Both the domain and application engineering levels of the SPL are specified from the perspective of the customer and other stakeholders involved in the problem space. At this phase, variability models are used to model the domain abstraction, while the application models represent the different configuration requirements.

On the other hand, the solution space perspective defines and develops the artifacts that will be used to implement the models defined in the problem space. These artifacts show the perspective from the SPL engineers' vision and may include components, libraries, classes, functions, methods, source code, configuration, and integration files, among other artifacts.

Both the problem and solution spaces, as well as domain and application engineering levels, are presented throughout the entire life cycle of the SPL processes. However, for the scope and purpose of this study we will exclusively focus on implementing our WUI Modeling Language at the domain engineering level.

BENEFITS AND CHALLENGES OF SPLs

One of the key factors of implementing software product lines is reusability. Software components reusability within a SPL can reduce the possibility of errors and bugs in a software system. As Pohl, Böckle, & Linden (2005) argued, by reusing components across different products, a SPL can avoid an over-engineering process for each product while reducing the number of defects in production. Furthermore, SPLs can enable better testability of software, as changes made to a common feature can be easily incorporated across multiple derived components.

Another benefit of SPLs is reduced time-to-market. By reusing software common features, software development teams can increase the development of new software components. SPLs can reduce the release time by providing a way to rapidly scaffold new products using existing software commonalities. This can be especially beneficial in industries with rapid technological change or a tight production roadmap.

However, the use of SPLs also poses several challenges. One such challenge is the need for a specific set of skills and tools. As Wąsowski and Berger (2023) assert, the process of implementing a SPLs requires a specialized set of skills and tools which can lead to create some entry barriers when lacking from these specialized technologies. To that point, VariaMos Web application represents a breakthrough by providing a solid framework and the tools required for modeling and reasoning on SPLs, context-aware systems, cyber-physical systems and WUIs.

COMPOSITIONALITY OF SOFTWARE SYSTEMS

Compositionality is a fundamental concept in mathematics and specifically in Applied Category Theory (ACT). Compositionality refers to the property of a system to be composed of smaller parts that can be combined in a predictable way to form a larger

system (Fong & Spivak, 2018). ACT has provided a powerful framework for studying compositionality in various fields, including chemistry, neuroscience, systems biology, natural language processing, dynamic systems, and database theory (Bradley 2018). There are also some initiatives to apply compositional properties in SPL engineering and software product families as in the studies conducted by Bosch, (2007); Munoz, Gurov, Pinto, and Fuentes, (2021); Taentzer, Salay, Strüber and Chechik, (2017).

One of the main benefits of compositionality relies on preserving systems consistency, allowing the creation of complex structures from underlying system building blocks. Compositionality is particularly important in software engineering as it enables the capabilities to manage complexity within the design of Systems of Systems (SoS).

Therefore, the use of compositionality in systems design and engineering has been the subject of extensive research in recent years. For instance, Schweiker et al., 2015 demonstrated how compositionality is a fundamental design technique for modeling distributed systems that can be easily and progressively changed over time. Likewise, the same authors proposed a formal framework based on ACT for reasoning about compositional SoS, demonstrating how these techniques can be applied to various engineering problems.

Another important application of compositionality in computing programming is the notion of *composition of functions* typically used within a statically-typed functional language. Typed Functional Programming (FP) is based on a rigorous mathematical foundation providing a powerful approach for designing functional programs that can be easily analyzed, extended, and modified over time (Hindley & Seldin, 2008; Motara, 2020, 2001).

> ***Note.*** The composition of functions is a fundamental topic in this study as the WUIs here introduced will entirely rely on modeling components in a functional style. In other words, we will use functions to give our code a place to live and types as the vehicle to compose them.

To summarize, compositionality is a fundamental concept in mathematics and computer science that has important applications in systems design, software

engineering, and computer programming. The use of compositionality enables the implementation of composable software systems that can be easily modified and extended. As software systems continue to grow in complexity, the use of compositionality will become increasingly important for modeling complex systems and Systems of Systems. In the subsequent chapters, we will demonstrate how both variability and compositionality complement each other seamlessly to model a WUI at a domain engineering level.

CHAPTER 3. PROBLEM STATEMENT

DESCRIPTION OF THE PROBLEM

Currently, WUI development processes are Web and product design focused. Although Web design is intended to improve user experience and product design, these methodologies lack an exhaustive engineering process to manage and extend WUI capabilities at scale. Traditional Web design and development processes have proven to work appropriately for specific tailor-made customizations but lacking from a modeling and engineering process.

We highly emphasize on understanding and differentiating designing from modeling, as this distinction often blurs when referring to WUIs. In that regard, we consider that having a modeling language tool is aimless if there is no opportunity to exploit it. In other software engineering processes, it is widely accepted to use modeling tools. However, in the context of web design and development, this practice has been ignored or not sufficiently explored. To some extent, this is related to the absence of WUI modeling tools and the misconception of using them.

To tackle these drawbacks, we have implemented a WUI Modeling Language that enables the incorporation of both variability and compositionality techniques. Thus, to leverage the WUI modeling process our proposal will rely on incorporating a SPL framework, at a domain engineering level.

RESEARCH QUESTIONS

- To what extent can WUIs be modeled to demonstrate variability properties?
- To what level can functional components be composed to obtain a WUI model at the domain engineering level?
- To what level can VariaMos framework and Web toolkit facilitate the design and implementation of a WUI Modeling Language?
- How feasible is it to incorporate a modeling stage into the process of designing and developing a WUI?

HYPOTHESIS

By implementing a WUI Modeling Language within VariaMos framework and Web toolkit, we will demonstrate to what extent both variability and compositionality techniques can be effectively integrated for modeling WUI at a domain engineering level.

CHAPTER 4. RESEARCH METHOD

DESIGN SCIENCE RESEARCH METHODOLOGY

To answer the formulated research questions and test the hypothesis, this study is seamlessly aligned with the *Design Science Research Methodology (DSRM)* by Peffers, Tuunanen, Rothenberger, and Chatterjee, (2007). The DSRM methodology consists of six main steps, related among them as presented in *Figure 3:*

1. Identify Problem and Motivate
2. Define Objectives of a Solution
3. Design and Development
4. Demonstration
5. Evaluation
6. Communication

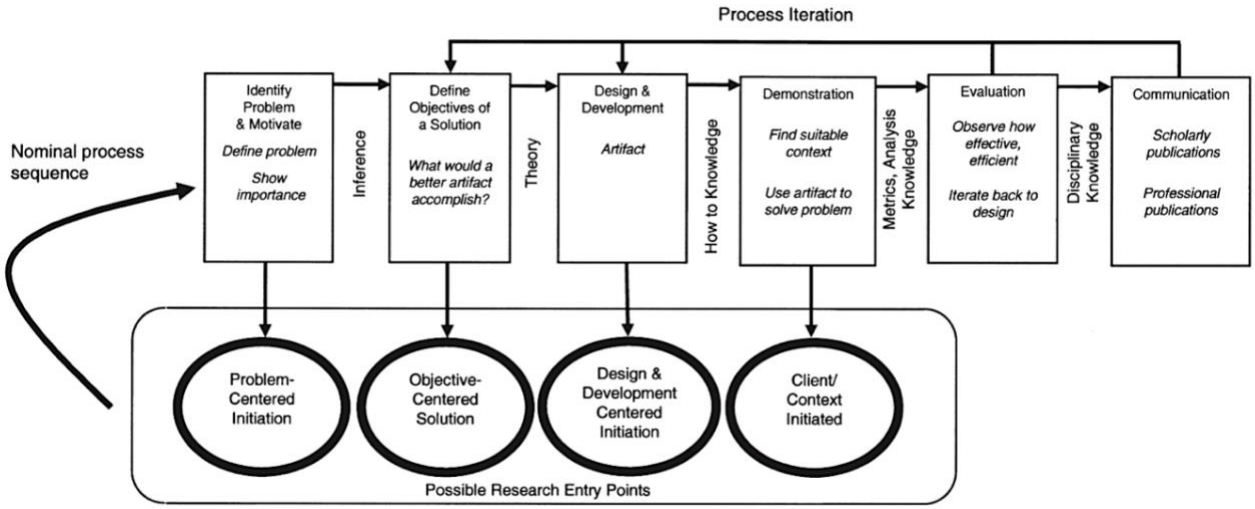

Figure 3. DSRM Process Model

By following this methodology, we can guarantee and acknowledge the design, implementation, and installment of the WUI Modeling Language within VariaMos framework.

Each of the aforementioned steps is explicitly framed within the organization of our study. By following the proposed design framework judiciously, we ensure this work to be highly structured and clearly detailed.

CHAPTER 5. LITERATURE REVIEW

WUIs VARIABILITY

Web User Interfaces (WUIs) have become ubiquitous components in computing, and their design and development has been subject of extensive research and technological advancements. As we have previously introduced, a WUI is a type of user interface that allows Human-Computer Interaction using all kinds of graphical elements on customized and personalized layouts Weld et al., (2003). Within the *Software Development Lifecycle (SDLC),* the design and implementation of a WUI can significantly impact the overall development process in terms of quality, cost, and time-to-market.

WUI implementation processes combined with variability techniques appears as a promising intersection to model complex user interfaces at scale. Consider, for example Pleuss, Hauptmann, Keunecke, and Botterweck, (2012) case study on variability in user interfaces. Similarly, different researchers and practitioners have explored how variability management can be applied to WUI modeling processes by refining the components reusability as well as supporting dynamic and self-adaptive behaviors (Nilsson, Floch, Hallsteinsen, & Stav, 2006). However, as Kramer, Oussena, Komisarczuk, & Clark (2013) suggested, still “there is little known how to handle user interface variability statically or dynamically” (p. 25).

Specifically, WUIs variability refers to a modeling technique that emphasizes the modularization, management, and reuse of user interface components across different products, SPLs, or software-product families. WUIs variability allows programmers to define a set of core features and functionalities that are shared across all the derived WUI products (Pleuss et al., 2012).

One way of applying SPLs to WUI development is by using feature models. As we have previously introduced, feature models are a modeling technique that allows programmers to capture the variability of a software product line in terms of its features and their dependencies. In the context of WUI development, feature models can be used to define a set of features that can be enabled or disabled based on the specific requirements of each product or product family. By defining these features and their dependencies, programmers can create a configurable and extensible WUI architecture that can be easily adapted, re-configured, and re-combined (Stuerzlinger, Chapuis, Phillips, & Roussel, 2006).

In fact, an intuitive approach for implementing a SPL into a product containing a user interface is to model one single SPL that covers all software layers, including the UI (see for instance, UI Catalog feature in *Figure 4*). This implies treating the WUI as an equally important asset of the SPL like other components of the application (Pleuss, Hauptmann, Dhungana, & Botterweck, 2012).

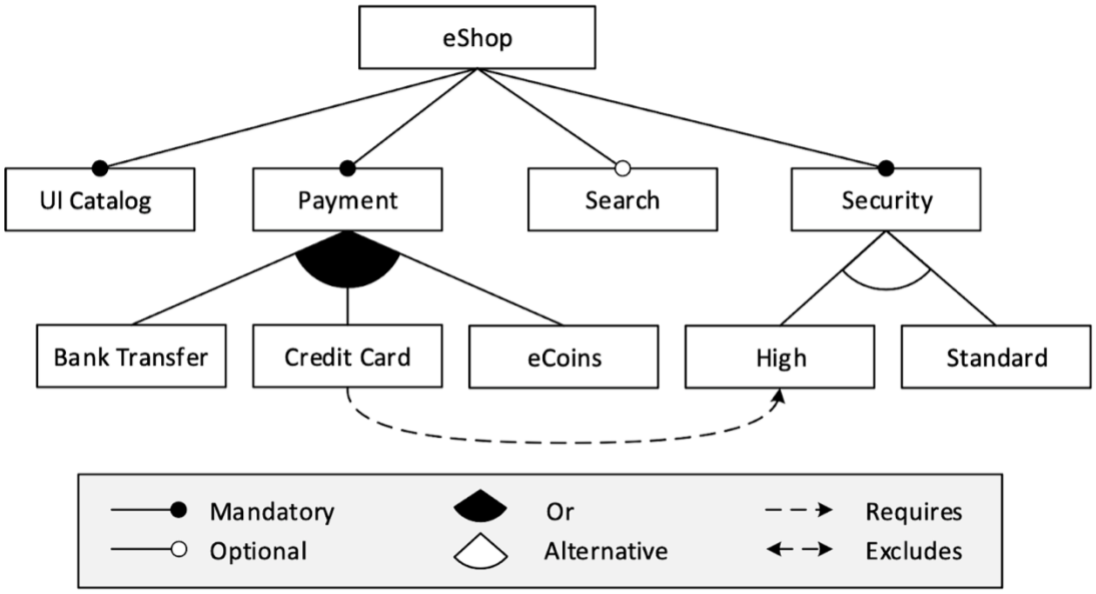

Figure 4. A Feature Model for an eShop

However, considering the advancement of more complex, robust, and interactive user interfaces, it has become more compelling to develop a stand-alone Software Product Line (SPL) for the user interface to cover all the specific constraints. Consider for instance the study conducted by Häuslschmid, Bengler, and Olaverri-Monreal, (2013) on the adaptive layouts of an in-vehicle user interface.

Another approach to WUI development within SPLs is to use a Component-Based Development (CBD). This approach emphasizes the design and construction of software systems using a set of independent, reusable, and interchangeable components. In the case of WUI development, CBD is applied by defining a set of WUI components which are reusable across different products or product families. CBD appears as suitable complement for variability and reusability process as well as perfect WUI modeling mechanism, “as many modern development and UI frameworks follow a component-based UI development style that promotes the decomposition of UIs into individual components” (Yigitbas et al., 2019, p. 2).

Variability models can take on many representation forms, from a mere informal diagram to a well-defined formal model that can be used to manage variability and automated derivation of products (Bachmann, 2004). Overall, variability models have the

advantage of making variability explicit and easy to configure. We believe that it should be used as a valuable artifact in the software engineering process (Geertsema & Jansen, 2010).

WUIs COMPOSITIONALITY

The design and development of user interfaces has always been a challenging task especially when it comes to modeling and implementing complex WUI with many components and lots of moving parts. One approach that has gained popularity is the use of compositionality, which involves separating the interface into smaller components that can be nested and composed. We explored the use of compositionality in the design and development of WUIs by introducing some basic concepts from Functional Programming (FP) and language type systems.

Similarly to a complex system, WUI composition emphasizes on breaking down the interface into smaller components or WUI building blocks that can be composed together to form a complete self-contained new system. In this regard, there are a few conducted studies in the composition of WUIs (Finnie, 1998; Hanus & Kluß, 2008), and the process and tools required to implement WUIs compositionality has been little explored.

Even though compositionality has been used in various ways in the system design. One technique is to use functional languages and statically type system mechanisms to compose WUI components that can be reused and combined in different ways (Achten, Eekelen, & Plasmeijer, 2004). Likewise, Banavar (1995) has introduced the notion of *compositional nesting,* allowing a module to be nested based on compositional operations. Compositional nesting entails the same meaning of an *operadic* structure as shown in *Figure 5* (Schweiker et al., 2015). These structures will signify the backbones of our WUI Modeling Language as they perfectly illustrate how components can be nested to form larger structures.

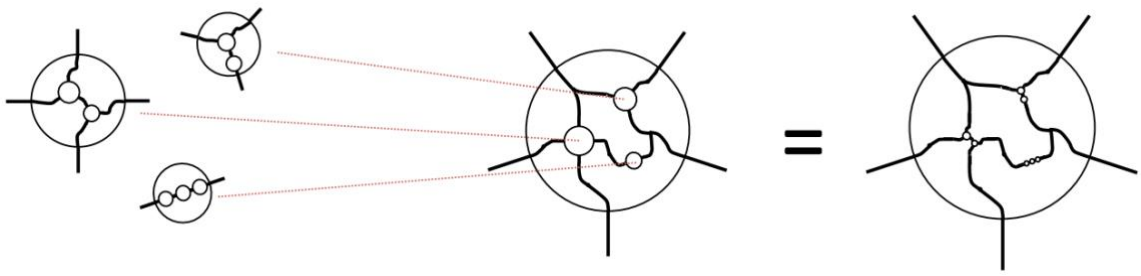

Figure 5. Operadic Structure

Graphical representation of an operadic structure as collection of nested operads that allow for complex composition of systems at different granularity levels.

As we have explored, compositionality is a powerful technique to model user interfaces since it allows programmers to break down complex interfaces into smaller and more manageable components. Along with the language type system, a declarative design supports optimal evaluation strategies, new design patterns, and it leads to better abstractions in application programs and within the Web (Antoy & Hanus, 2022). So, a declarative design consists in the definition of functions and data types on which the functions operate, allowing functions to become first-class citizens to fully exploit the power of logic programming (Hanus & Kluß, 2008). Thus, depending on the programming language specification, today it is possible to use a range of different languages on the client side. Consider for example: *JavaScript, TypeScript, Scala.js, Elm, PureScript, and ReasonML.* One main advantage of using these technologies for implementing (WUI) in a functional programming style is that these languages support standard features like *Higher-Order Functions (HOFs).* A HOF is a function that takes one or more functions as arguments and return a function as their result. In the context of modeling and implementing a WUI in a functional style, we could create a HOF that takes a functional component as an argument and returns a new component that includes the performed functionality.

On the other hand, using statically-typed programming languages brings the ability to combine types. In the majority of statically-typed languages, the built-in data types can be combined to create custom types, which help on implementing type-safe and

expressive programs. Another advantage of using statically-typed programming languages is the ability to leverage types for compositionality purposes. By using types to represent the structure and behavior of software components, programmers can implement compositional building blocks that can be combined in various ways to create larger systems (Hanus, 2006; 2007).

To sum up, working with types and functions as fundamental compositional structures has been a technique explored for increasing the modularity and composition of software systems. Conducted research on the topic of compositionality has demonstrated the advantages of types and functions which make this a suitable complement for implementing our WUI models. Thus, compositional techniques are particularly useful in complex WUIs where interfaces may have several components and many moving pieces to be managed. Additionally, compositionality helps to separate concerns making it easier to test and debug individual software parts.

Overall, the use of compositionality in the modeling of user interfaces is a valuable tool for software engineers to create effective and efficient WUIs as we will demonstrate afterwards.

CHAPTER 6. TRADITIONAL DESIGN AND DEVELOPMENT PROCESS

WUI DESIGN PROCESS

At a high level, the traditional process of designing and implementing a user interface can be divided into two main process or stages[5] as shown in *Figure 6*.

[5] At a detailed level, the design stage typically comprises three main steps: *sketching, wireframing,* and *prototyping.* We are not focusing on them, but they are very important steps for ensuring that the final interface is well-designed, user-friendly, and meets the needs of the target audience.

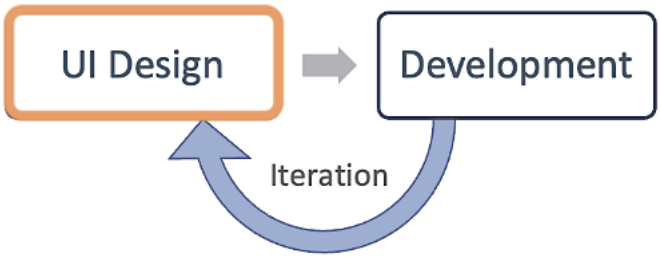

Figure 6. Traditional UI Design Process

In the design stage, designers focus on creating a visually appealing and user-friendly interface. This involves understanding the needs and preferences of the target audience, selecting appropriate color palettes, fonts, and graphics, as well as sketching and prototyping the overall components and layout within a design system. At this point, it is important to note that in many cases, programmers and software engineers are not included or interested in participating in this stage. This can be a drawback as the final design may not be aligned with what is possible to implement based on actual design requirements and the technical constraints. Therefore, it is crucial for developers to be involved earlier in the design stage so they can provide opportune feedback on what is feasible to implement and suggest alternative solutions when necessary.

The design stage may involve several iterations to refine the interface and make it as user-friendly and effective as possible. However, due to time constraints or the need to share the design with clients or stakeholders, it's not always possible to spend as much time as desired on this stage. As a result, designers may need to make decisions about the interface based on incomplete information or without the benefit of extensive testing and feedback.

Once the UI prototype and mockup are done, they are passed to the development team who are responsible for implementing the interface.

WUI DEVELOPMENT PROCESS

Within a traditional design and development process, the next step is to implement the actual user interface. This involves writing the code that will make the design come to life and enable the interactive behaviors and functionality as depicted in *Figure 7*.

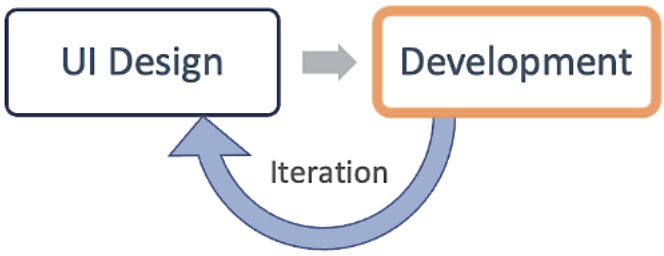

Figure 7. Traditional UI Development Process

The programming languages, libraries, frameworks, and tools used in this process may vary depending on the specific project, but common ones include *HTML, CSS, JavaScript, TypeScript, Angular, and React*.[6]

Here, it is important to note that sometimes the design and development process are not aligned, as there may be some designs that are not always achievable or practical to implement. This can be understood as the design and development processes often follow a waterfall model, where development can only begin after the UI prototype and mockup have been completed. This can lead to many drawbacks if there are design changes or unexpected technical constraints that arise during the development phase.

To improve this design and development process, a modeling stage has been proposed that involves incorporating a WUI model allowing for greater collaboration between designers and engineers.

[6] For a broader list of technologies redirect to *State of JavaScript* https://stateofjs.com

WUI MODELING PROCESS

This stage involves creating a model of the interface that can be specified, validated, and refined before the development stage begins. During this stage, programmers can ensure that the interface components are not only functional, but also manageable, maintainable, and easy to modify. Within a traditional process, this modeling stage can be incorporated within the overall design and implementation process as shown in *Figure 8*.

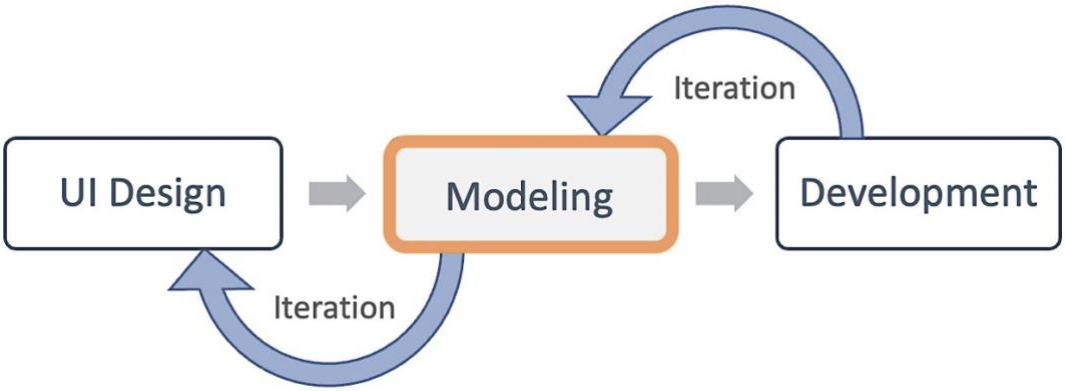

Figure 8. WUI Modeling Process

Consider this an iterative process, rather than a task of consecutive steps. Besides, this modeling stage does not aim at following a *Big Design Up Front (BDUF)*. On the contrary, this model phase and the resulting models are open to be initiated, accessed, and modified at any stage of the whole SDLC. That is why our modeling process will be better conceptualized within a SPL engineering framework.

CHAPTER 7. WUIs MODELING ENGINEERING

WUI DOMAIN ENGINEERING PROCESS

As we have previously explored, both variability and compositionality techniques have become an integral part of modern software engineering for modeling user interfaces. However, there is not a consensus on what it means to model a WUI and how

it differs from the Web design process. Furthermore, based on the literature review conducted there are a few tools for implementing these languages and models. Lastly, both variability and compositionality techniques have been individually studied for modeling a WUI but not incorporated together. Consequently, our contribution relies on the following improvement we will develop along the subsequent chapters: (a) incorporating a modeling stage at a domain engineering level, (b) implementing a WUI Modeling Language, and (c) combining both variability and compositionality within one single model and tool.

In the context of this research, domain engineering involves analyzing the domain of a WUI and its variability models. This process generates a set of artifacts that will be used in subsequent levels as shown in the SPLE framework. Common and variable elements of our WUI model are defined using variability and compositionality techniques. Thus, we have broken down this modeling stage into three main steps: *(a) componentization, (c) variability,* and *(d) compositionality.*

- **Componentization:** This step involves breaking down the user interface into individual components or elements such as buttons, forms, navigation menus, etc. These components are identified and defined based on their purpose, function, and design. Each component is treated as a separate entity that can be reused across the whole system. WUI components are self-contained, meaning they can be easily added, removed, or modified without affecting the other components. At this stage, we can also consider a modularization step to organize the individual components into larger, functional modules. However, for simplicity, consistency, and standardization, we have decided to refer to any WUI element as a component. Accordingly, we will refer to features of the WUI, such as login, checkout, catalog, as WUI components.
- **Variability:** This step involves identifying the different components that exhibit any variability property based on different criteria, such as user role, customization and criticality. It is understood that not all components or properties of a system can be considered suitable for variability. Therefore, this is an extremely important step in which it is determined which components may be an option for variability, without

affecting the normal functioning of the system. In this way, variability in our model will be configured at two levels: boolean variability and property-level variability.

- **Compositionality:** This step involves defining how components can be composed to create more complex user interface structures and preserve the system. For this step, it is important to have previously defined the components properties and the possible nominees for variability. Without completing these previous steps, the use of composition will not have much impact. In that regard, compositionality will determine the mandatory or optional nature of the components and their properties. This will be accomplished through the use of statically-typed programming language, in our case TypeScript.

By incorporating these three modeling stages into the overall process, developers can implement interfaces that are not only accessible and visually appealing, but also functional and reliable. So, one of the advantages a modeling process and artifact brings up is the possibility to identify potential issues early on, during the implementation of the WUI, or even after the development process.

For future research on this topic, subsequent analysis should emphasize this modeling stage at the *application engineering level*. The application level would involve developing a WUI product according to specific requirements by reusing the components that have been designed and modeled in the *domain engineering phase*. Considering the scope of this research the *configuration* and *derivation* processes of a WUI product are not included in this study, although they will be the main topics to be developed in subsequent research projects.

CHAPTER 8. MODELING LANGUAGE PROPOSAL

WUI MODELING LANGUAGE

We propose the design and implementation of a modeling tool that makes use of VariaMos framework in order to incorporate a *WUI Modeling Language*. This language will enable the creation and exploration of different WUI use cases facilitating the management of variability and compositionality. The modeling language will be

implemented using VariaMos framework, by extending abstract and concrete syntax mechanisms.

VARIAMOS FRAMEWORK

VariaMos is a powerful framework that allows users to create and customize modeling languages and execute operations over the implemented models. VariaMos Web application provides dynamic support for defining modeling languages, design models, and reasoning on these models. VariaMos also offers various capabilities to support and extend syntax and semantics for different languages and programming paradigms. Additionally, it enables the definition of operations for different modeling languages, enabling users to create projects both at a domain and application engineering levels as shown in *Figure 9*.

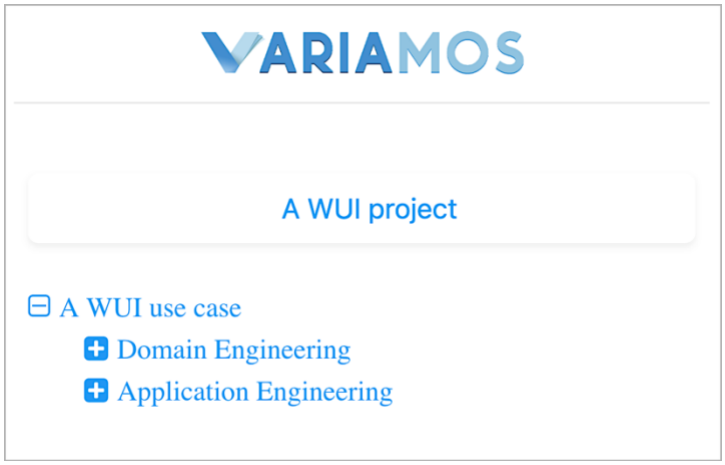

Figure 9. VariaMos Engineering Levels

The framework also allows for the dynamic extension of VariaMos to design new modeling languages as well as providing configurable solvers to execute the defined operations and constraints over the instances of these languages.

The language development capability of VariaMos is particularly noteworthy, as it supports the definition of both abstract and concrete syntax and semantics. This enables users to define and customize modeling languages based on their specific needs and

requirements. Moreover, the framework is continuously updated in order to support new modeling capabilities, such as the definition for different design extensions, specifications, and constraints.[7]

CHAPTER 9. MODELING LANGUAGE DESIGN AND IMPLEMENTATION

EXTENDING VARIAMOS SYNTAX

One advantage of using VariaMos framework is the possibility of configuring and extending its syntax and semantics. For our specific case, we will only focus on syntax configuration. For future research, we will be explored to incorporate semantic configurations to our language. The syntax configuration of our models is a process that can be carried out from the platform itself, as shown in *Figure 10*. For this, we go to the project management section and select settings. We see two configuration options: *(a) abstract syntax* and *(b) concrete syntax*.

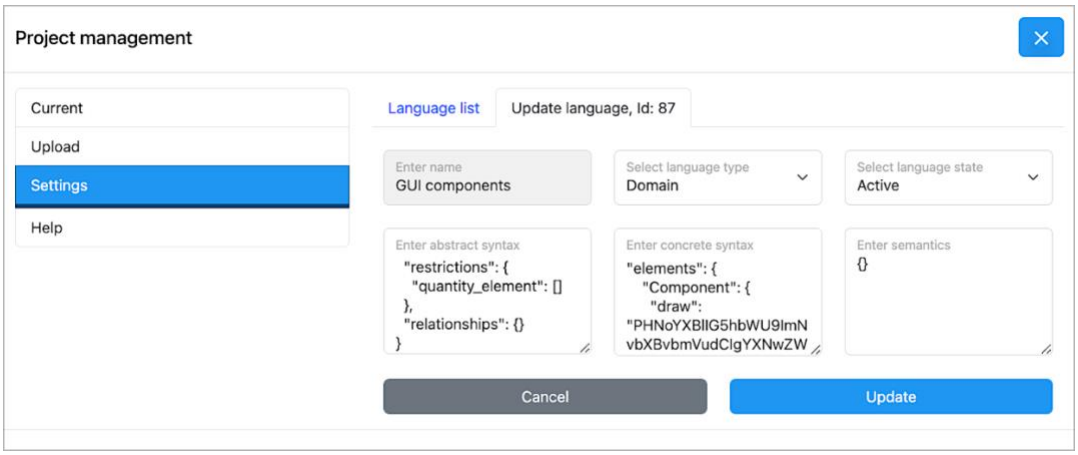

Figure 10. VariaMos Abstract and Concrete Syntax

[7] For more information on *VariaMos* framework, Web application, and modeling tools redirect to https://variamos.com

- **Abstract Syntax:** The abstract syntax of a model is the set of relationships and constraints of the language, according to its characteristics. In the case of our language, the abstract syntax represents each of the variability and composition configurations of our language. These configurations include both the variability mechanisms and the composition specifications.
- **Concrete Syntax:** Regarding the abstract syntax of a model, it contains all the visual elements of the language. For our specific case, we have configured this syntax according to each of the diagrammatic elements that will abstractly represent the components and types, and relationships of our model.

Both the abstract and concrete syntax extension of the variability and compositionality models represent the design foundation of the WUI Modeling Language. In the subsequent sections, we will present some of the specific features of the VariaMos environment while also introducing some of the basic elements of the implemented modeling language.

WUI MODELING LANGUAGE BASICS

For the implementation of the WUI Modeling Language, we have divided the domain engineering workspace into two main models: *(a) WUI Components Model* and *(b) WUI Model* (see *Figure 11*).

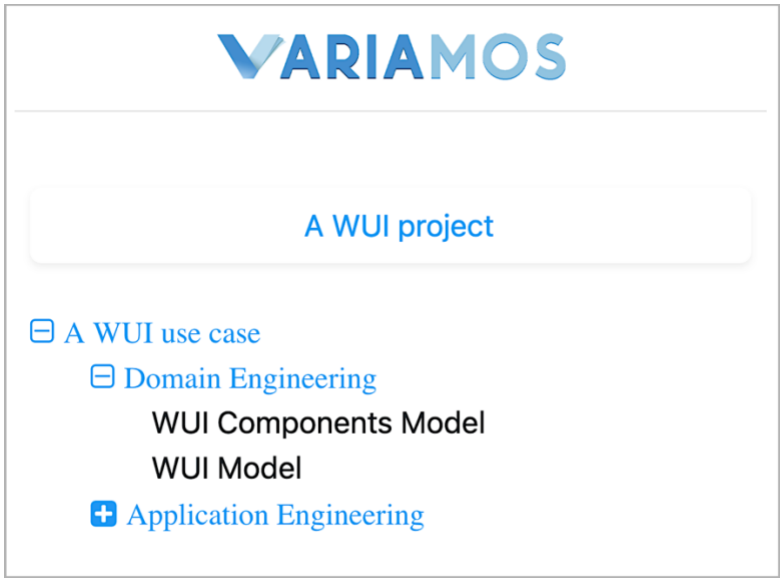

Figure 11. WUI Domain Engineering Models

- **WUI Components Model:** In this model workspace, the initial creation of components will be carried out, exploiting their nesting capabilities.
- **WUI Model:** In this model workspace, the previously created components in the component model will be instantiated. In addition, this will be the space to initiate the process of configuring the variability and composition of our use cases.

Once a project has been initialized at the two model workspaces mentioned earlier, we can begin to design and implement our use cases. Typically, modeling languages designed using VariaMos Web application and tools are based on a series of language-specific elements. These elements can be visualized in the Elements menu located on the right-hand side (see *Figure 12*).

Depending on the characteristics of the language, it may have one or many elements that will serve as abstractions in the models or diagrams. In our specific case, our modeling language will only have a single element, "component," in this initial part.

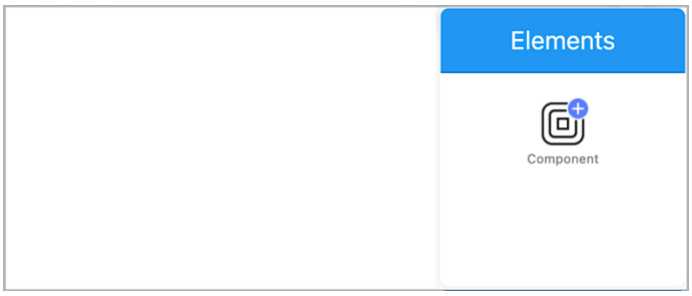

Figure 12. Elements Menu

A basic step-by-step process to create a component in our language involves dragging the component from the elements menu and dropping it onto the diagram editor (see *Figure 13*). Immediately, a box will be displayed, which will represent one of our component's model at a high-level of abstraction.

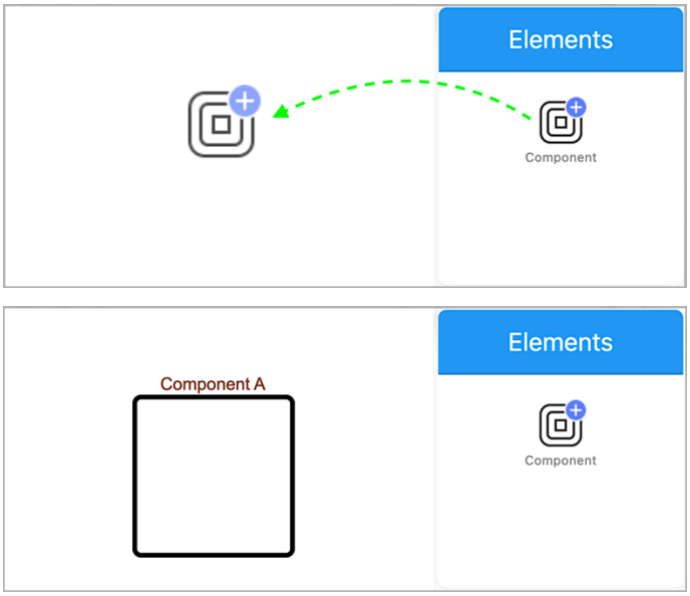

Figure 13. How to Create a Component

Once we have our first component, we can nest another component inside the initial component to create a slightly more elaborate component. To do so, we repeat the previous step, but this time making sure that the component is properly nested. In this case, the diagram will display a blue box indicating that the components can be nested, as exemplified in *Figure 14*.

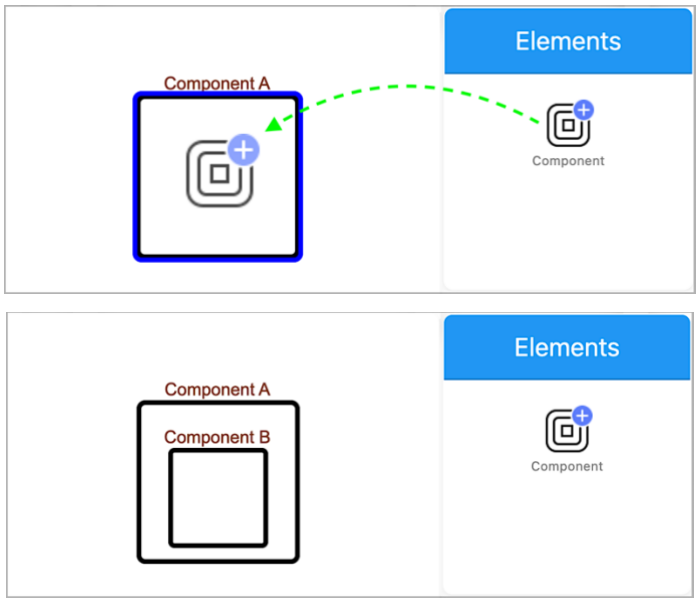

Figure 14. How to Nest a Component

To change a component position, it just requires moving the component from one place to another. However, what we want to emphasize here is the ability of a component to preserve its internal structure and arrangement. As we can see in *Figure 15,* both Component A and Component B maintained their nesting relationship.

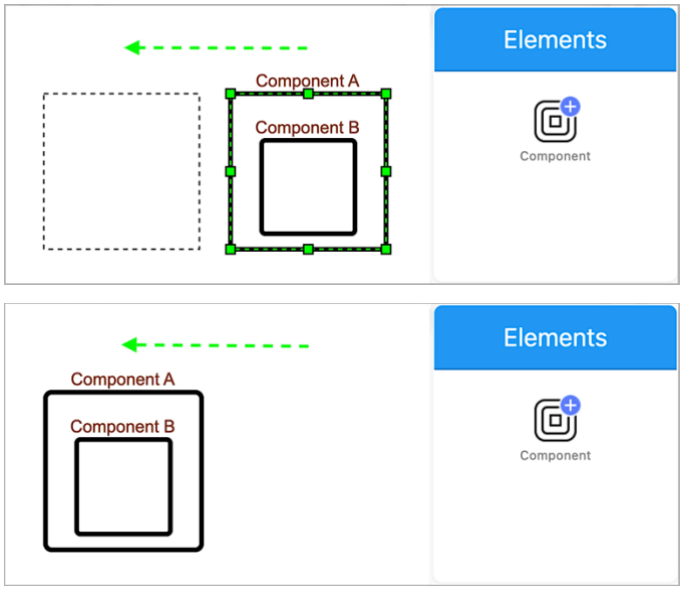

Figure 15. Nested Component Move and Arrangement

Lastly, the WUI Model will have three additional elements in addition to the component element. These elements are the *m-Type* and *o-Type* data types, and a third element that represents the instantiation of the components created in the WUI Component Model, as presented in *Figure 16*. We will elaborate on these elements more in depth when we address the sections on variability and compositional mechanisms of our models.

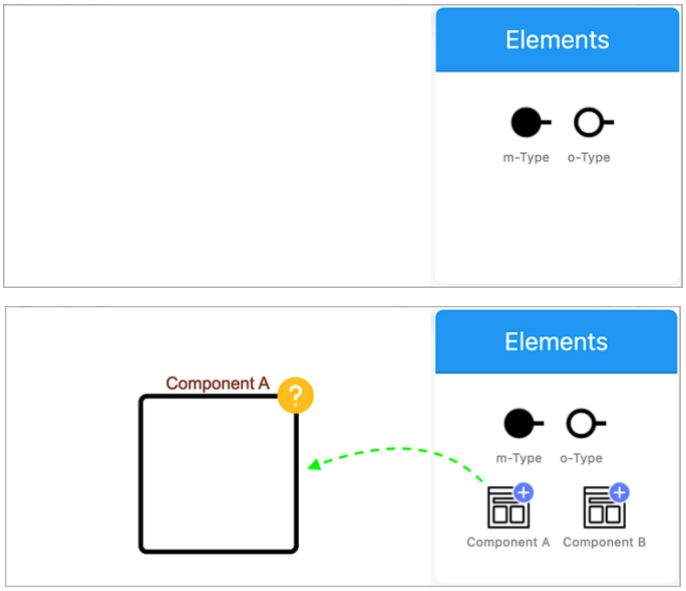

Figure 16. Components Reusability

Thus, the number of dynamically instantiated elements in the WUI Model will depend on the number of components created in the WUI Components Model. Therefore, typically the element menu of the WUI Model will look like *Figure 17*.

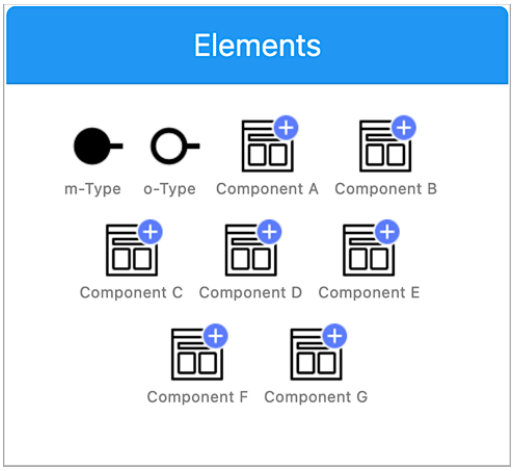

Figure 17. Dynamically Instantiated Components

Once we have understood the basic steps for creating a component, we will now proceed to describe in more detail its nesting structure capabilities.

WUI COMPONENTS MODEL

The WUI Components Model is an essential feature of our modeling language tool. This model is intended for the initial stage of the WUI modeling process where individual and generic components are modeled as the WUI building blocks. By representing the component in a nested diagram, our modeling language enables the zoom-in and out the component granularity level, making it easier for programmers to reason about structure and complexity of the WUI (see *Figure 18*). At this stage, each component is modeled as a self-contained unit that can be reused and composed with other components to form more complex components.

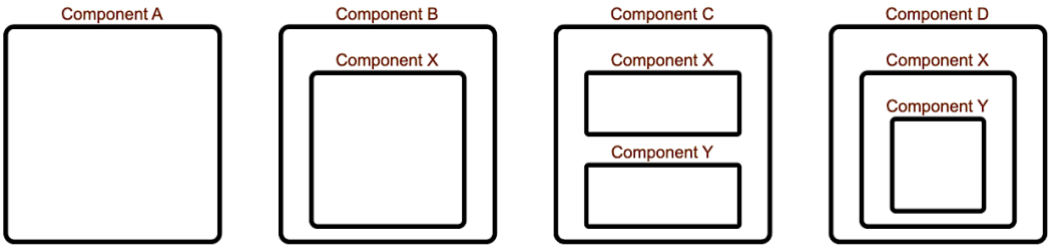

Figure 18. Components Model

The WUI Components Model is based on the concept of functional components, which are small, reusable units of design that can be used to build larger, more complex structures (see *Figure 19*). Each component is represented as a box of nested elements or components, which provides a visual representation of the WUI model structure. Likewise, the WUI Components Model includes the basic component properties and a textbox introducing the corresponding component code.

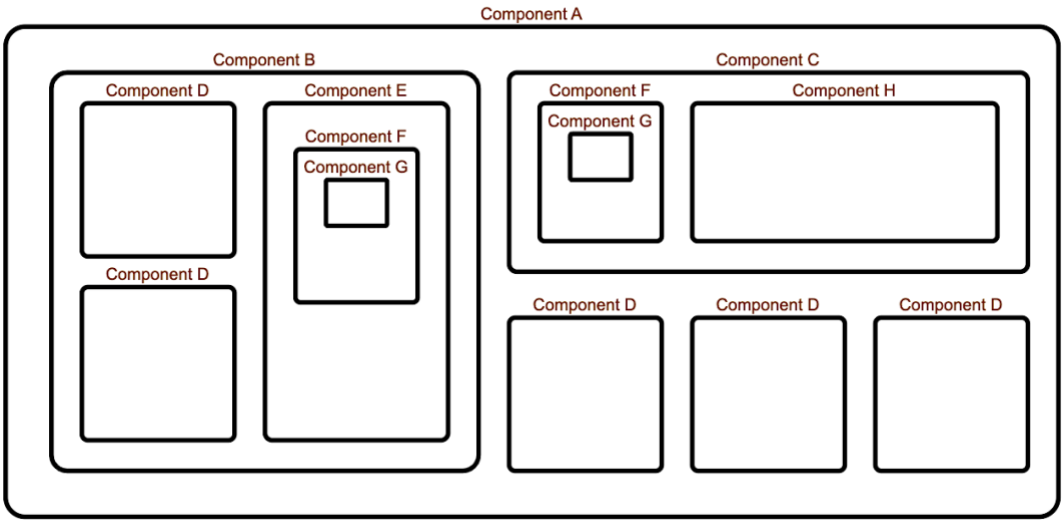

Figure 19. A Large Component's Model

The WUI Components Model is intended to promote components reusability and improve efficiency in the WUI modeling process. By breaking down a WUI into smaller, reusable components, designers and developers can easily create a component pool or library that can be used across multiple applications. This approach saves time and effort by eliminating the need to recreate components for each new project.

Each component in the WUI Components Model is designed to be self-contained and independent of other components. This means that components can be reused and composed with other components without affecting their behavior or composition. That is to say, each component is designed to be free from side effects, which are common sources of bugs and inconsistencies in software systems. By enforcing statically-typed functional components, the WUI Modeling Language ensures that each component is well-defined and can be safely composed in the subsequent steps.

WUI VARIABILITY MECHANISMS

To incorporate variability in our models, VariaMos allows for the configuration of *variability flags,* as shown in *Figure 20*.

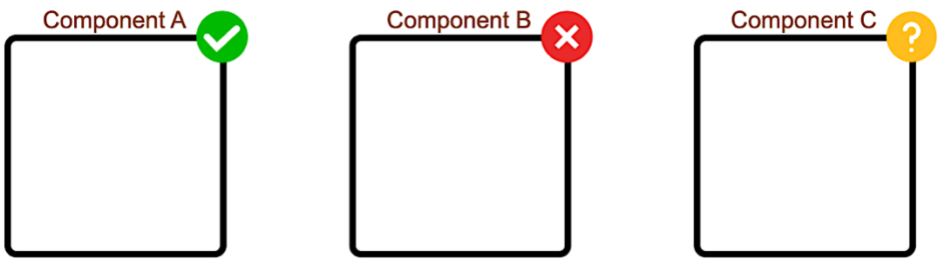

Figure 20. VariaMos Variability Flags

- **Green flag:** indicates that the component is an active part of the model and will be included at derivation stage.
- **Red flag:** states that the component is inactive and will not be part of the product once its derivation is completed.
- **Yellow flag:** specifies that the component is a derivation candidate, and the decision to include it or exclude it has not yet been defined. Notice, if a component is checked with the yellow flag, it also means that the component is inactive by default.

Based on the above mechanisms, we have identified two variability states:

- **Active Components:** green flag components included on derivation.
- **Inactive Components:** red flag component excluded from derivation, and yellow flag components variability candidates.

Afterwards, we will determine how these flags are the base of the *boolean variability* concept.

WUI MODEL

In addition to enabling the creation of the self-contained components in the previous stage, our WUI Modeling Language includes a WUI Variability process, which incorporates configurable features for variability management. Specifically, the WUI

Model addressed variability at three levels: *(a) Boolean-Level Variability, (b) Property-Level Variability,* and *(c) State-Level Variability.*

Boolean-Level Variability

The Boolean-Level Variability allows for the inclusion or exclusion of a WUI component or module based on a boolean restriction. This process enables the management of large WUI components and modules and the subsequent configuration and derivation at product line. For the purpose of this research, we will exclusively focus on how to manage variability at domain engineering level.

Property-Level Variability

The Property-Level Variability allows for the variation of a WUI customization property. This level of variability enables the creation of highly personalized components that can meet a wide range of user needs and requirements. For instance, a WUI component's font size, color scheme, and layout can be varied based on the preset configurable properties.

State-Level Variability

Similar to the boolean variability, the state variability allows to show or hide components behaviors but at a type property level.

DIAGRAMMATIC REASONING

The design of our WUI Modeling Language is founded in the notion of diagrammatic reasoning as a powerful technique for understanding complex systems and modeling their behavior. Thus, diagrammatic reasoning is the process of reasoning using modeling language, often referred to as formal models or diagrammatic syntax. Diagrammatic reasoning has been extensively used in the fields of mathematics, computer science, and engineering to represent complex systems and structures, and to derive insights from them.

In the context of formal modeling languages, diagrammatic reasoning refers to the use of diagrams to represent and reason about formal models. One example of a formal diagrammatic syntax are *string diagrams,* which are widely used in Applied Category Theory and typed Functional Programming to represent the composition of functions.

Thus, diagrammatic reasoning is a powerful tool for representing and reasoning about complex structures and systems in depth. As we have previously mentioned, in software engineering, diagrammatic reasoning and string diagrams can be used to represent and analyze the structure and behavior of software systems or Systems of Systems. By using formal diagrams to represent and reason about systems composition, we can gain a better understanding of their behavior and derive insights that would be difficult to obtain through other means.

STRING DIAGRAMS

String diagrams are important mathematical objects that we will incorporate as we advance in our study. They were introduced in the mathematical context, specifically in the context of Category Theory by Joyal and Street (1993), but they have been used by engineers and scientists in various contexts. String diagrams, also known as wiring diagrams, are visual representations for building new relationships or compositions, as in *Figure 21,* examples.

There are many styles and conventions for drawing string diagrams; however, the sort of diagram we will cover is that with input and output boxes composed in series from left to right.

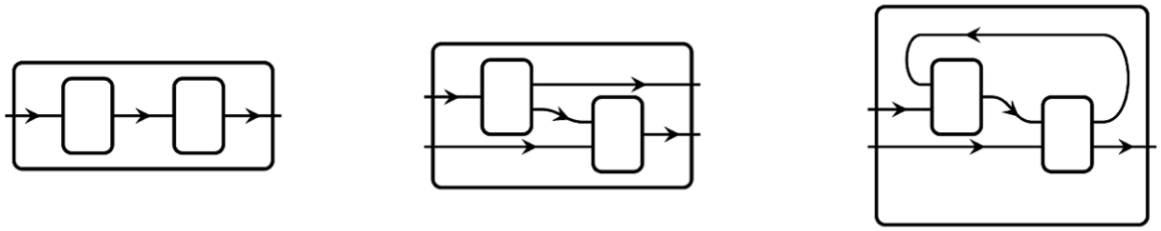

Figure 21. Wiring Diagram Example

We can have more complex string diagrams and more complex structures, including parallel composition, but these structures are not included in our models. In our case, WUI boxes will have multiple inputs and outputs, and they may be exclusively composed in series. In a string diagram the input represents elements, the boxes represent process, and the string diagrams themselves show how relationships can be combined. (Fong & Spivak, 2018). For example, given two functions *f and g,* their composite function *g* ○ *f* can be represented using a wiring diagram as the composition series of their individual objects, with the output of *f* being connected to the input of *g,* as shown in *Figure 22.* In other words, ○ composition symbol means *"do the processes in series."*

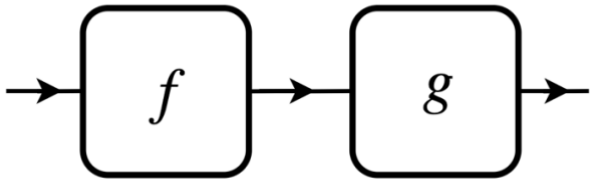

Figure 22. Composition of *g* ○ *f* read as *"g after f"*

In the context of our WUI models, String diagrams will provide a visual representation of the composition of functions. One of the key benefits of string diagrams is that they provide a way to reason about our WUI models composition based on types and functions, making it easier to reason about complex functions and to derive new functions from existing ones.

WUI COMPOSITIONALITY MECHANISMS

As we have previously proposed, a novel approach to model user interfaces is to use diagrammatic reasoning and string diagram mechanisms to enable the representation of the UI components by extending the TypeScript type system and functional capabilities. But how to represent that type system in our model? Similar to a feature model, we will borrow the concepts of *mandatory* and *optional* relationships for our models with some slight changes, as illustrated in *Figure 23*.

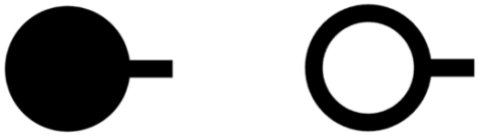

Figure 23. Mandatory and Optional Types

Specifically, we will refer to these relationships as mandatory and optional data types, so we came up with two configurable options:

- **m-Type:** stand for mandatory data type.
- **o-Type:** stand for optional data type.

The mandatory data types will represent the active components and the mandatory properties of our components, while the optional types are meant for setting up the variability of our components and properties. Notice that *m-Types* can exclusively be composed to *m-Types,* and *o-Types* exclusively to *o-Types*.

To incorporate a mandatory or optional type into our model, we navigate to the elements' menu, located on the right-hand side, where we will find both data types in a panel as the one presented in *Figure 24*.

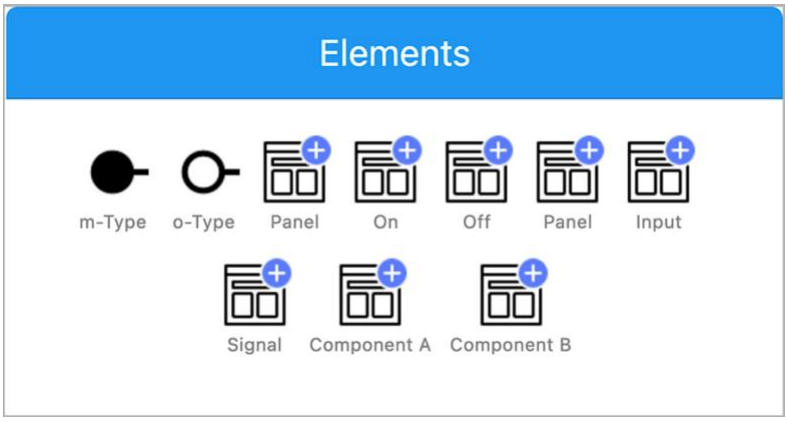

Figure 24. m-Type and o-Type in the Elements' Menu

To add a data type to our component, we drag the corresponding element and place it inside the component. In the case of data types, we place them on the left or right side of the component like *input* and *output* ports (see *Figure 25*).

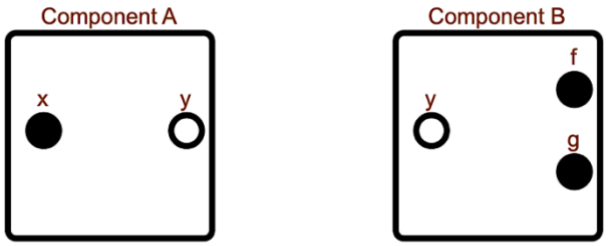

Figure 25. Various Types Inside a Component

A component can have as many data types as properties, so the limit will depend on the characteristics of the component. To compose a component, we place the cursor on the output type and drag a line to the input type (see *Figure 26*).

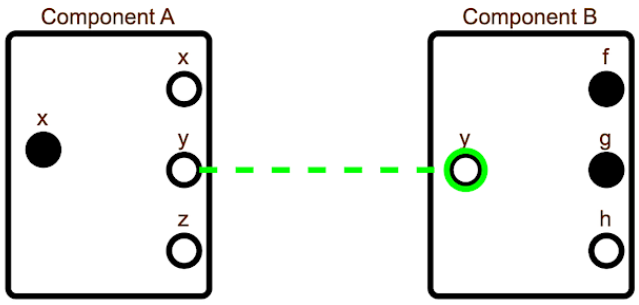

Figure 26. Components Composition Steps

Immediately, a dotted line will appear, indicating that the data types are the same, allowing the components to be composed. Then, an arrow will appear, indicating that the components effectively satisfy the composition laws (see *Figure 27*).

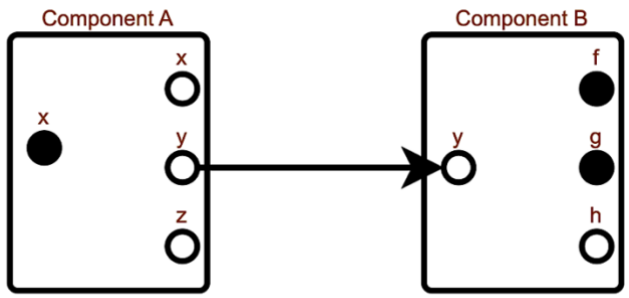

Figure 27. Components Composition

However, if for some reason we mistakenly decided to connect two components that do not satisfy the rules of composition by data types, the model will not allow it in any case. Even if both are o-Types as shown in *Figure 28*. In this regard, the composition rules must be satisfied in both cases, at the m-Type and o-Type restrictions and at the property data type level.

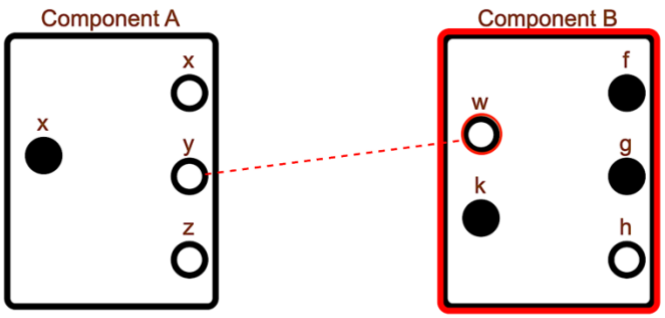

Figure 28. Type Composition not Satisfied

This is a simple but powerful condition established in the nature of string diagrams. Unlike other models, to satisfy the composition laws of a system, data types must be included first. This simple assertion is what makes string diagrams a very powerful tool for modeling critical software systems. In our case, the data types constraints will not only let us configure components variability but also manage complexity.

On the other hand, one of the advantages of modeling our system based on types and functions is that they are highly compatible with functional programming languages based on Static type checking. For our case, TypeScript has been adopted as our implementation technology. However, it does not mean a model can be technology agnostic. On the contrary, other functional scripting languages such as Scala.js, Elm, and PureScript could also be used.

In addition to the variability levels, the WUI Model also provides compositional-based configurations and abstractions. As a matter of fact, we have also incorporated required and optional constraints, similar to a feature model but at a type level composition.

Mandatory Composition

The mandatory composition is represented with a black filled circle type, and it means that any components requiring or exposing a mandatory property type must be composed to satisfy the model's law.

Optional Composition

On the contrary, the optional composition is represented with a white circle type, and it represents the component's properties or data types that are not obligatory to be composed or triggered.

> ***Note.*** Both the mandatory and optional composition aim at safeguarding the model from any inconsistency at the variability level. Since changing, modifying, or removing any piece within our model can lead to software malfunctioning, type safety guarantees the model consistency.

Thus, TypeScript static type checking will allow us to catch errors and prevent bugs by ensuring that the components within our model are correctly composed and behave as intended.

Overall, the WUI Model is a powerful feature of the modeling language tool that enables developers to build complex and adaptable UIs by reusing and composing functional components based on a statically-typed language system. This enhances the scalability, maintainability, and reliability of the UIs, making it easier for developers to build and maintain large-scale UI applications.

TYPESCRIPT OVERVIEW

The use of TypeScript for Web development has gained popularity due to its ability to provide type checking, type systems expressiveness, and safety. TypeScript provides a JavaScript superset that allows programmers to write code that is easier to understand and maintain. Therefore, by reasoning in terms of TypeScript type system and functional style, we can leverage our WUI Modeling Language to implement more expressive and powerful structures.

Contrary to C# and Java Object-Oriented Programming (OOP) languages, where classes are the basic unit of code organization, TypeScript uses functions to give the code a place to live. While the usage of classes is also permitted in TypeScript, it has become less common recently. Forcing all behaviors and data to be held within a class can be the right way to model some problems, but not every domain has to be represented this way. Additionally, in JavaScript, *free* functions can be instantiated anywhere, and data can be passed around freely without being inside a predefined structure. While using a dynamic type system and free functions can provide greater flexibility, it can also introduce flaws and errors into the system. Therefore, many programmers have opted to use TypeScript as the predetermined tool based on its type system capabilities and the possibility to be incorporated within a functional programming style.[8]

[8] Redirect to this link for the *TypeScript* official documentation https://www.typescriptlang.org

CHAPTER 10. MODELING LANGUAGE USE CASES

CREATING A WUI PROJECT

A WUIs Modeling Language has been designed and implemented making use of VariaMos, abstract and concrete syntax capabilities, and by leveraging the power of diagrammatic reasoning. Our modeling language tool provides the expressiveness for modeling WUIs based on the three modeling steps proposed early. By representing these compositions as string diagrams, our modeling language tool enables the easy zoom-in and zoom-out of the component granularity level, making it easier for programmers to reason about the behavior and structure of the WUIs. Next, we will demonstrate the capabilities of the WUI Modeling Language implemented by exemplifying the characteristics addressed in this study through a use case.

MODELING A CONTROL PANEL WUI

To demonstrate the WUI Modeling Language usage, a basic *Control Panel WUI* has been chosen as our use case. A Control Panel WUI consists of various components, which can be composed and reused to form more complex and customizable systems. Using the WUI Componentization Model, each individual component of the Control Panel WUI has been modeled and defined with their properties, interfaces, and code. At the highest level of abstraction, our first approach to modeling the basic control panel components would look like this, see *Figure 29*.

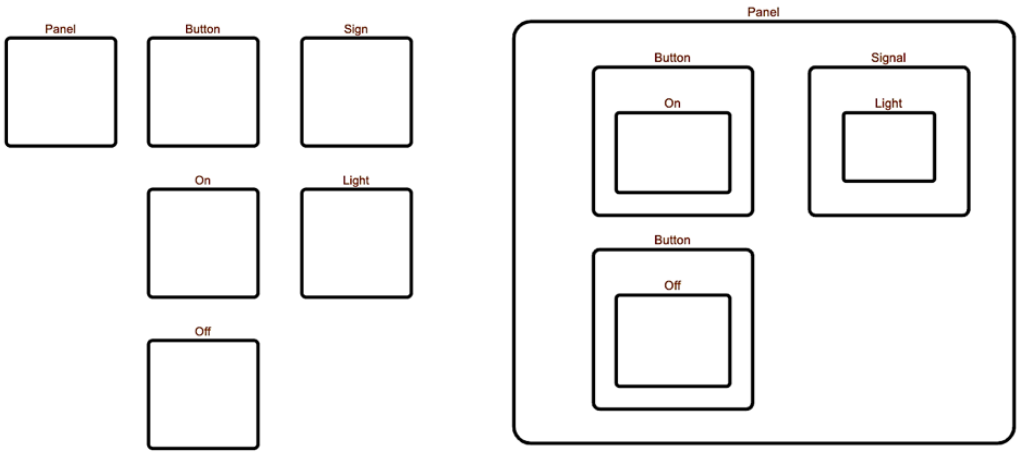

Figure 29. High-Level Control Panel Components

Once we have preliminarily modeled the basic components that will be part of our Control Panel, we will move on to the next modeling stage. For this, we go to our WUI Model. In this workspace, we will see that the components that we had previously modeled in the WUI Component Model are now instantiated in the element menu in order to be reused (see *Figure 30*)

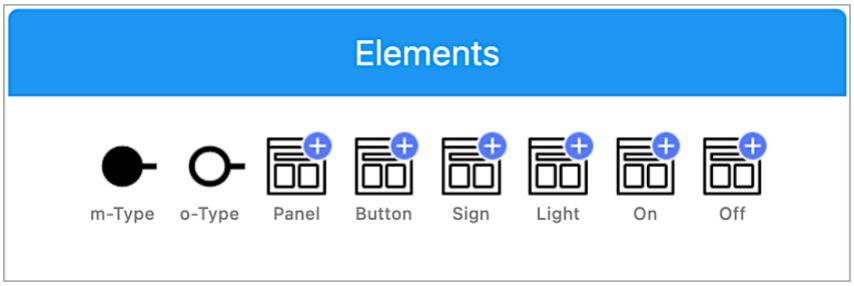

Figure 30. Instantiated Control Panel Components

Next, we will proceed to configure the variability of our control panel. To do this, we must determine which components are variability prone. We can make this configuration using variability flags as we already explain. For our use case, we have determined that the Sign component that emits light is not necessarily required if both the On and Off buttons provide feedback to the user. Similarly, to proceed with the

composition process, we add data types according to the properties of each component and its variability constraints (see *Figure 31*).

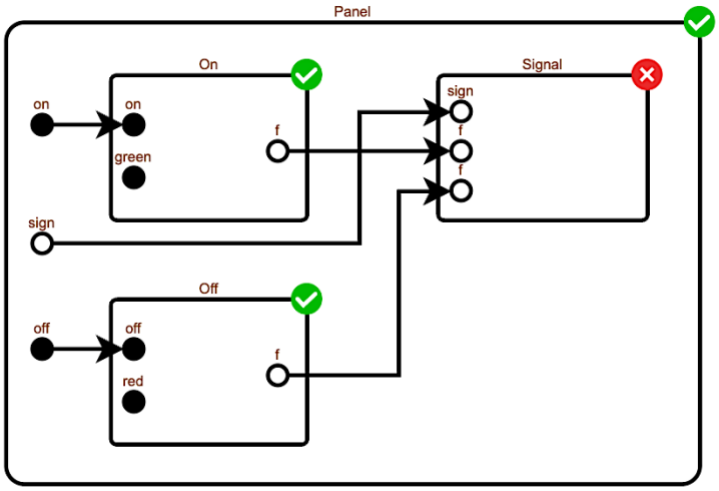

Figure 31. A Control Panel WUI

If we have followed configuring steps for componentization, variability, and compositionality, we have successfully modeled our first use case using the WUI Modeling Language.

To conclude, we will demonstrate some of the most important features of our model. Like string diagrams, our model has the ability to be reasoned in detail within its internal components. Therefore, if we zoom-in on one of the buttons on our control panel, we can see its internal components as presented in *Figure 32*.

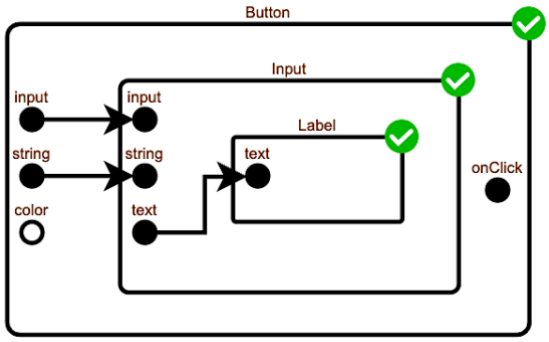

Figure 32. Control Panel Button Zoom-In

Furthermore, if we navigate to the properties' sidebar panel, we can see the TypeScript code of our button component. Here it is important to notice the expressiveness of our model. As we can see, the properties of our component correspond exactly to the types instantiated in the code as presented in *Figure 33*.

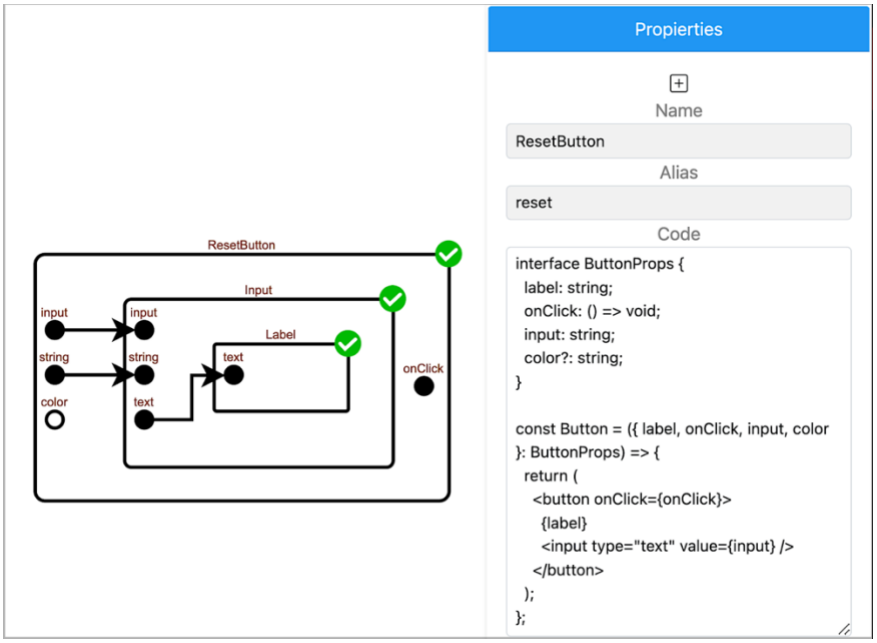

Figure 33. Components' Properties and Source Code

Lastly, if instead we decide to zoom-out, we will likely find that the Control Panel is part of a much larger system, such as a dashboard, as exemplified in *Figure 34*. However, this is not a restriction since our control panel was built based on the composition of types and functions which will preserve our system configuration conditions.

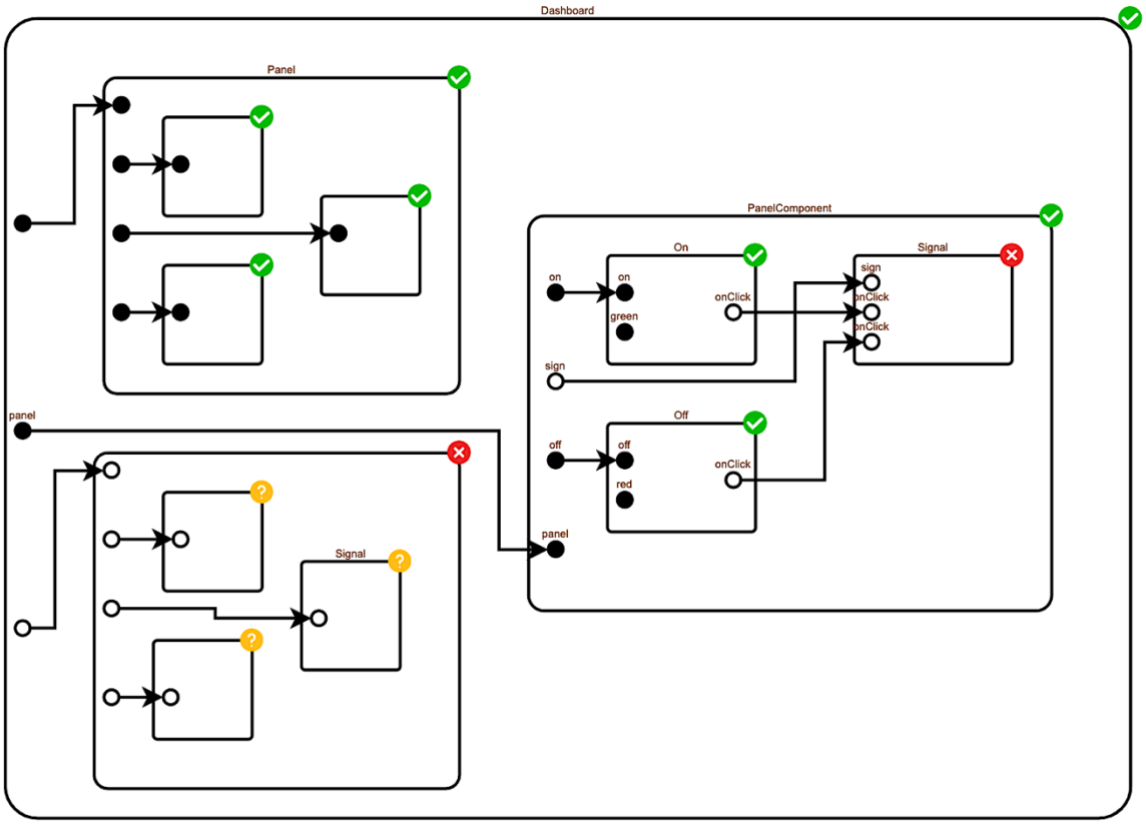

Figure 34. A Dashboard WUI Zoom-Out

To conclude, our WUI Modeling Language will enable programmers to create variable components that can adapt to different user requirements, such as different display layouts or functionalities. By using Boolean-Level Variability and Property-Level Variability, the Control Panel WUI can be customized based on user preferences and needs, without the need of additional constraints configurations.

Likewise, the WUIs Compositionality Model will allow developers to compose the components to form more complex and functional structures. By representing these compositions as a string diagram, the tool enables easy visualization and manipulation of the component, enabling programmers to reason about the behavior and structure of complex WUIs.

As we have shown with this example, the WUI Modeling Language offers a powerful set of modeling capabilities for modeling complex and adaptable WUIs, as

demonstrated through this use case. Finally, by using Typescript and string diagrams, the tool enables programmers to build scalable, maintainable, and reliable WUI systems, reducing the time and effort required for software development and maintenance.

CHAPTER 11. EVALUATING AND VALIDATING A WUI MODEL

MAINTAINABILITY QUALITY ATTRIBUTE

Maintainability represents the degree of effectiveness and efficiency to which a product or system can be modified, improved, corrected, or adapted. We consider maintainability as the software quality attribute that is better aligned with our use cases. Thus, we have designed a maintainability test to be executed in the modeling phase assuming later we will implement the WUI. This test can be compared to an *Architecture Tradeoff Analysis Method (ATAM),* in the sense that both an ATAM and the maintainability test we are about to administer are analysis conducted prior to the implementation of the actual software system.[9]

DATA GATHERING

For data gathering, validation, and evaluation of a model, we conducted a test to assess the maintainability of a use case developed in a WUI Modeling Language. Specifically, we chose the Control Panel as the model for analysis. A digital test was administered to a sample of 13 software developers divided into two teams and distributed as following:

- **Team 1:** Consisting of seven developers.
- **Team 2:** Consisting of six developers.

[9] For further information on the *maintainability* attribute redirect to https://iso25000.com

All developers are part of the same project within a software development company in the private business sector.

Mostly, development teams are typically composed of a maximum of 5 to 9 developers, depending on the project's characteristics. Besides, maintaining compact teams allows for more efficient communication flow. Several development teams typically constitute a software project within a company.

For our specific case, both teams are part of the same project, which develops custom software products at both the presentation layer and the domain level. Both teams consist of at least 4 or 5 developers, a web designer, and a project manager. Due to confidentiality reasons, none of the respondents requested to have their names disclosed.

For the purposes of confidentiality and internal policies, both the company and the developers entered into a confidentiality agreement, whereby any data or related information from this study should not be shared through any physical or digital means.

MAINTAINABILITY TEST

This test aims to evaluate the following criteria: *modularity, reusability, analysability, modifiability,* and *testability* based on a Control Panel WUI model from *Figure 35*.

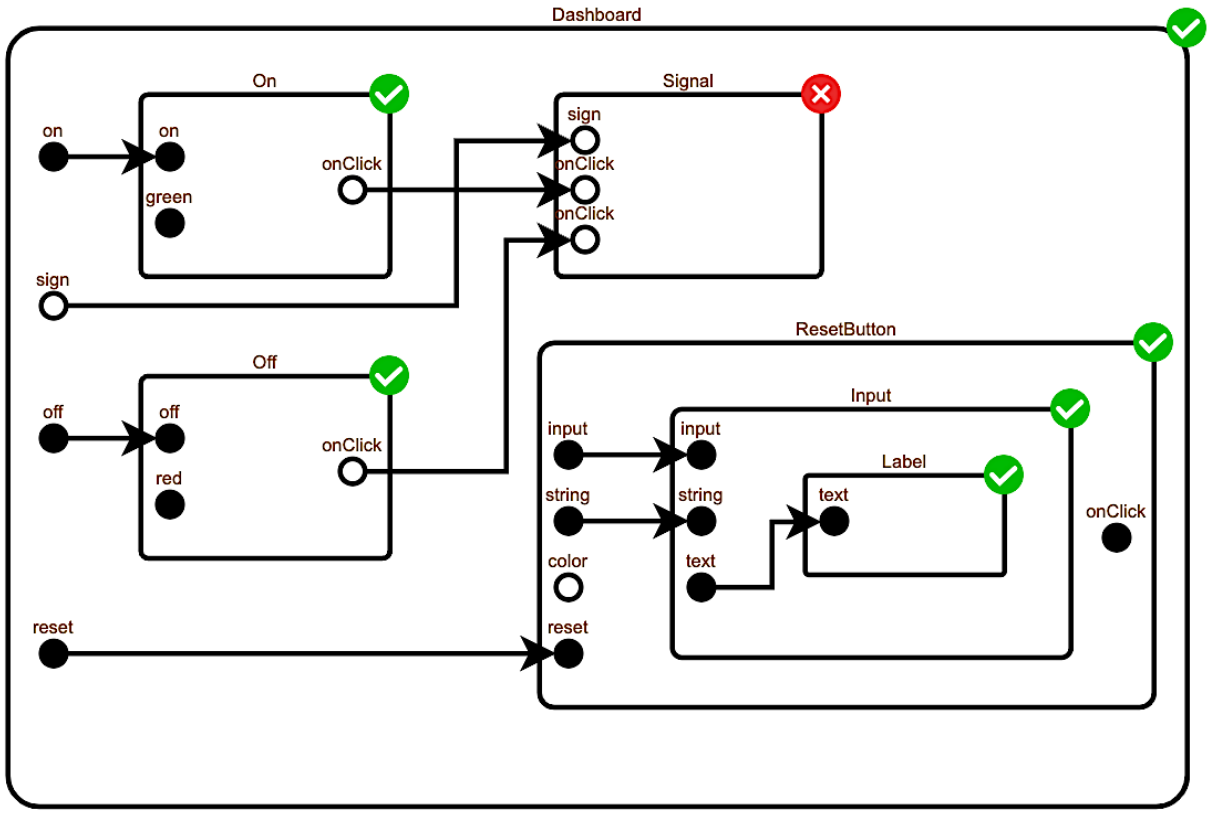

Figure 35. Control Panel for Maintainability Test

Guidelines: Answer the following questions on a range level from *1 to 5,* meaning *1 the model does not satisfy the requirement criteria,* and *5 the model fulfills the requirement criteria.* Provide any additional comment if the model successfully accomplishes the criteria, or on the contrary you might find any inconsistency regarding the requirements.

Table 1. Maintainability Questionnaire

Name: _____ ***Date:*** _____

Role: _____

Company: _____

Requirement	Criteria	Level	Additional Comments
Modularity	To what level the WUI model is composed of discrete components or modules such that a change to one component has minimal impact on other components?		
Reusability	To what level the WUI components or modules can be used in more than one system, or in building other components or assets?		
Analyzability	To what level is it possible to assess the impact on the WUI model of an intended change to one or more of its parts, or to diagnose within the model for deficiencies or causes of failures, or to identify parts to be modified?		
Modifiability	To what level the WUI model can be effectively and efficiently modified without introducing defects or degrading existing components or the model quality?		
Testability	To what level test criteria can be established for the WUI model, components or modules and tests can be performed to determine whether those criteria have been met?		

Confidential Agreement: By signing this form, you are declaring your data to be confidential and that it should not be shared through any means. _____

FINDINGS

Here is the final result and consolidated data from the conducted test. The data is presented according to each team. For quantitative measurement, we have calculated the average of each attribute and its standard deviation. We have also calculated the overall average of both teams to finally present the consolidated comments for each item.

Table 2. Modularity Level Results

To the question *"to what level the WUI model is composed of discrete components such that a change to one component has minimal impact on other components?"*

Team 1 (Seven Developers)	**Team 2** (Six Developers)
• Two developers rated the model as 5 • Four developers rated the model as 4 • One developer rated the model as 3	• Three developers rated the model as 5 • Three developers rated the model as 4
Modularity average level 4,15 Standard deviation 0.69	**Modularity average level 4.5** Standard deviation 0.55
Overall modularity average level 4.3	

Consolidate Comments

- *"Components somehow evidence the modularity of the UI"*
- *"Components are well defined"*
- *"There is not a clear differentiation between a small component and a container"*
- *"Is the whole Control Panel a component by itself?"*
- *"It would be nice to differentiate presentation components for logical components (components with a behavior)."*

Table 3. Reusability Level Results

To the question *"to what level the WUI components can be used in more than one system, or in building other components or assets?"*

Team 1	Team 2
• One developer rated the model 5 • Three developers rated the model 4 • Two developers rated the model as 3 • One developer rated the model as 2	• One developer rated the model as 5 • Four developers rated the model as 4 • One developer rated the model as 2
Reusability average level 3.58 Standard deviation 0.98	**Reusability average level 3.83** Standard deviation 0.99
Overall reusability average level 3.71	

Consolidate Comments

- *"The components can be reused exclusively into this system, but it is not clear to what extent they can be reused in other products."*
- *"The Control Panel components can be reused but they need to be adjusted to the target system."*
- *"Sometimes it could be easier to develop the components from scratch than reusing the existing ones."*

Table 4. Analysability Level Results

To the question *"to what level is it possible to assess the impact on the WUI model of an intended change to one or more of its parts, or to diagnose within the model for deficiencies or causes of failures, or to identify parts to be modified?"*

Team 1	Team 2
• One developer rated it as 5 • Two developers rated the model as 4 • Three developers rated it as 3 • One developer rated the model 2	• One developer rated the model 5 • Two developers rated the model as 4 • Three developers rated it as 3
Analysability average level 3.43 Standard deviation 0.98	**Analysability average level 3.67** Standard deviation 0.82
Overall analysability average level 3.55	

Consolidate Comments

- *"At a high level of abstraction, the model is self-explanatory."*
- *"The model needs a chart with the name of each element in the diagram."*
- *"It is possible to assess the model because the ports safeguard the system."*
- *"Although the type system helps on the WUI model analysis, they just represent a tiny fraction of the system, and not the UI as a whole."*

Table 5. Modifiability Level Results

To the question *"to what level the WUI model can be effectively and efficiently modified without introducing defects or degrading existing components or the model quality?"*

Team 1	Team 2
• Three developers rated the model as 4 • Two developers rated the model as 3 • Two developers rated the model as 2	• One developer rated the model as 5 • One developer rated the model as 4 • Four developers rated the model as 3
Modifiability average level 3.14 Standard deviation 0.9	**Modifiability average level 3.5** Standard deviation 0.84
Overall modifiability average level 3.32	

Consolidate Comments

- *"The type system mechanisms easily allow the modification of the system."*
- *"The model allows identifying which elements can be modified, extended or excluded because of the mandatory and optional configurations."*

Table 6. Testability Level Results

To the question, *"To what level test criteria can be established for the WUI model, components or modules and tests can be performed to determine whether those criteria have been met?"*

Team 1	**Team 2**
• One developer rated the model as 3 • Four developers rated the model as 2 • Two developers rated the model as 1	• Three developers rated the model as 2 • Three developers rated the model as 1
Testability average level 1.89 Standard deviation 0.69	**Testability average level 1.5** Standard deviation 0.54
Overall testability average level 1.68	

Consolidate Comments

- *"It is not possible to test a system that has not been implemented."*
- *"It isn't clear how to apply this attribute into the model."*
- *"The model can help to prevent and identify incidents in advance, but not in testing the system."*

DATA ANALYSIS

Overall, the data gathered shows a homogeneous pattern in both teams according to the average of each attribute. Additionally, the standard deviation of the sample is consistent with low disparity according to the responses range. On the other hand, the comments positively help to better understand the quantitative data for each of the criteria assessed.

For the modularity item, both teams agreed that the components do express the system's modality, although the type of components are not clearly specified. Regarding reusability, it is not clear to what extent a component can be reused in another system without the need to modify or implement it again. Regarding the analysability of the model, the possibility of analyzing the model is expressed, but only at a high-level of abstraction since the model only shows some UI elements but not the entire conglomerate of internal components. Concerning the modifiability of the system, the advantages of working with a strongly typed system are expressed to configure restrictions on components. Finally, for the testability item, the inability of the model to establish testing cases is strongly expressed both in the comments and in the average rate.

LIMITATIONS AND OPPORTUNITIES

These data will allow us to identify improvements in the tool and how models should be shared. In this particular case, participants only had access to a static image or model that did not allow them to interact with it. This is one of the most relevant items as it will allow us to consider ways in which models can be shared to enable interaction with them, as well as to limit or configure editing settings. Regarding this item, the implementation of collaborative tools within VariaMos Web application has been already identified. This will allow not only collaborative access to the tool, but also extend its use in the software industry.

On the other hand, in evaluating the models with the previously provided test, our aim was to demonstrate the validity of the tool in a real context and use case. Indirectly, if the analysis of a real use case is supported by the tool, this would provide an initial approach to its validation. For this study, however, external validity cases involving the direct use of the tool were not considered.

To make our modeling language available to the industry, the tool must first undergo a rigorous internal evaluation process that involves not only a conceptual evaluation but also usability tests. This is presented as one of the limitations and opportunities for an exhaustive validation of the tool, with the aim of incorporating this and other improvements in future studies.

Finally, our modeling language provides an initial approach to incorporating WUI models at the domain engineering level, but we are aware of the opportunity to complete the process at the product engineering level. For the sake of specificity and depth in the study, we consider it relevant to address both levels individually. Moreover, further research should be focused not only on the configuration and derivation of product lines, but also to the incorporation of code generation mechanisms, as well as the exploration of self-adaptable and dynamic product WUI systems.

CHAPTER 12. FINAL CONSIDERATIONS

FURTHER RESEARCH

Considering previous research constrains and opportunities, future studies could focus on exploring the usability of the WUI Modeling Language for implementing more complex WUI product lines, as well as investigating the impact of using SPLs on software quality metrics such as reliability, performance, and security.

Another challenge in the use of SPLs is ensuring that a common architecture is adaptable to meet the specific needs of different customer segments or markets. Thus, designing a common architecture that is both flexible enough to manage the variability and support the development of multiple products can be the focus for further studies.

Likewise, further studies can be focused on modeling other applications different than Web User Interfaces. For example, researchers could explore how to apply these techniques in the context of mobile or cross-platform applications. These applications have different constraints and requirements compared to Web applications, and thus, require a different design and modeling approach to be explored.

Lastly, further research is needed to study the effectiveness of the WUI Modeling Language in a real-world setting.

CONCLUSIONS

This research has contributed to the field of software engineering by demonstrating the feasibility of using SPLs methods and tools for modeling WUIs variability and compositionality at a domain engineering level. The implementation of a WUI Modeling Language within the VariaMos framework and Web tool has shown that it is possible to extend WUIs capabilities beyond a single custom product and achieve a higher degree of standardization in the software development process.

SPLs are a powerful approach to software engineering that can provide numerous benefits to WUIs modeling and design, including to improve the software quality, reduced time-to-market, and decrease development costs. However, this approach also poses several challenges, including the need for specialized skills and tools and the complexity of managing variability and compositionality. To address these challenges, this study substantially revealed how software development teams and organizations can take several steps, such as incorporating a modeling process, adopting the use of specialized tools, and incorporating variability and compositionality techniques. As WUIs continue to evolve, these steps will become more important. As we have also evidenced, managing variability in SPLs can be a challenging task, especially when it comes to ensuring that the different configurations of the system can be composed in a consistent and reliable way. This is where compositionality comes into play, as it enables programmers to implement reliable software systems by relying on formal specifications.

Finally, the incorporation of a SPLE framework into the WUI modeling process has shown promising results in terms of managing variability. As we evidenced on this study, SPLs allow programmers to implement configurable and extensible components that can be customized and adapted to different contexts. SPLs can achieve this by defining a set of common features, which can be reused across different WUI applications, and a set of variable features that can be customized based on specific requirements. Hence, both variability and compositionality techniques were used to break down the WUI models into smaller, more manageable components that were composed together by demonstrating how these techniques successfully increased type safety, modularity, and maintainability of a WUI use case.

REFERENCES

Achten, P., van Eekelen, M., & Plasmeijer, R. (2004). *Compositional Model-Views with Generic Graphical User Interfaces.* In: Jayaraman, B. (eds) Practical Aspects of Declarative Languages. PADL 2004. Lecture Notes in Computer Science, vol 3057. Springer, Berlin, Heidelberg. doi: https://doi.org/10.1007/978-3-540-24836-1_4

Antoy, S. & Hanus, M. (2002). *Functional Logic Design Patterns.* In: Hu, Z., Rodríguez-Artalejo, M. (eds.) *FLOPS 2002. LNCS, vol. 2441, pp. 67–87.* Springer, Heidelberg.

Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B., & Vilbig, A. (2004) *A meta-model for representing variability in product family development.* In Software Product-Family Engineering, pp 66–80

Banavar., G. S. (1995). An Application Framework for Compositional Modularity. *Department of Computer Science.* The University of Utah. Retrieved from https://www-old.cs.utah.edu/docs/techreports/1995/pdf/CSTD-95-011.pdf

Bačíková, M., and Porubän, J. (2012). Analyzing stereotypes of creating graphical user interfaces. *Cent. Eur. J. Comp. Sci. 2,* 300–315. doi: https://doi.org/10.2478/s13537-012-0020-x

Bosch, J. (2007). *Software Product Families: Towards Compositionality.* In: Dwyer, M.B., Lopes, A. (eds) Fundamental Approaches to Software Engineering. FASE 2007. Lecture Notes in Computer Science, vol 4422. Springer, Berlin, Heidelberg. doi: https://doi.org/10.1007/978-3-540-71289-3_1

Bradley, T. (2018). What is Applied Category Theory? *arXiv preprint.* Retrieved from https://arxiv.org/abs/1809.05923

Chen, X., He, J., Liu, Z., Zhan, N. (2007). *A Model of Component-Based Programming.* In: Arbab, F., Sirjani, M. (eds) International Symposium on Fundamentals of Software Engineering. FSEN 2007. Lecture Notes in Computer Science, vol 4767. Springer, Berlin, Heidelberg. doi: https://doi.org/10.1007/978-3-540-75698-9_13

Davies, J., Gibbons, J., Milward, D., & Welch, J. (2012). *Compositionality and Refinement in Model-Driven Engineering.* In: Gheyi, R., Naumann, D. (eds) Formal Methods: Foundations and Applications. SBMF 2012. Lecture Notes in Computer Science, vol 7498. Springer, Berlin, Heidelberg. doi: https://doi.org/10.1007/978-3-642-33296-8_9

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, A. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021). *Software Engineering Institute,* Carnegie Mellon University. Retrieved from http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

Kramer, D., Oussena, S., Komisarczuk, P., & Clark, T. (2013). "Graphical user interfaces in dynamic software product lines," 2013 4th International Workshop on Product LinE Approaches in Software Engineering (PLEASE), pp. 25-28, doi: https://doi.org/10.1109/PLEASE.2013.6608659

Finnie, S. O. (1998). Composing Graphical User Interfaces in a Purely Functional Language. Department of Computer Science, University of Glasgow Retrieved from https://theses.gla.ac.uk/1597/1/1998finniephd.pdf

Fong, B. & Spivak, D. I. (2018). An Invitation to Applied Category Theory: Seven Sketches in Compositionality. Cambridge University Press. doi: https://doi.org/10.1017/9781108668804

Geertsema, B. & Jansen, S. (2010). Increasing Software Product Reusability and Variability using Active Components: A Software Product Line Infrastructure. doi: https://doi.org/10.1145/1842752.1842814

Hanus, M. (2006). Type-Oriented Construction of Web User Interfaces. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2006), pp. 27–38. ACM Press, New York (2006)

Hanus, M. (2007). Putting Declarative Programming into the Web: Translating Curry to JavaScript. In: Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2007), pp. 155–166. ACM Press, New York

Hanus, M. & Kluß, C. (2008). Declarative Programming of User Interfaces. In: Gill, A., Swift, T. (eds) Practical Aspects of Declarative Languages. PADL 2009. *Lecture Notes in Computer Science,* vol 5418. Springer, Berlin, Heidelberg. doi: https://doi.org/10.1007/978-3-540-92995-6_2

Häuslschmid, R., Bengler, K., Olaverri-Monreal., C. (2013). Graphic Toolkit for Adaptive Layouts in In-Vehicle User Interfaces. doi: https://doi.org/10.1145/2516540.2516580

Hefley, W. E. & Murray, D. (1993) Intelligent user interfaces. doi: https://doi.org/10.1145/169891.169892

Hindley, J. R. & Seldin, J. P (2008). Lambda-calculus and combinators, an introduction. ISBN-13. 978-0521898850

Krishnaswami, N. & Benton N. (2011). A Semantic Model for Graphical User Interfaces doi: https://doi.org/10.1145/2034574.2034782

Joyal A. & Street R. “Braided tensor categories”. In: Advances in Mathematics 102.1 (1993), pp. 20–78 (cit. on pp. 40, 145).

Kumar, R., Natarajan, S., Shariff, M., & Mani, P. (2021). Dynamic User Interface Composition. In: Singh, S.K., Roy, P., Raman, B., Nagabhushan, P. (eds) Computer Vision and Image Processing. CVIP 2020. Communications in Computer and Information Science, vol 1377. Springer, Singapore. doi: https://doi.org/10.1007/978-981-16-1092-9_18

Linton, M., Vlissides, J., & Calder, P. (1989). Composing user interfaces with InterViews. doi: https://doi.org/10.1109/2.19829

LiQun Shan, Wei Wei, and Yanchang Liu (2010). Customizable WEB UI of Based on Templates. Information Technology Journal, 9: 1677-1681. doi: https://doi.org/10.3923/itj.2010.1677.1681

Nilsson, E. G., Floch, J., Hallsteinsen, S., and Stav, E. (2006). Model-based user interface adaptation DOI: https://doi.org/10.1016/j.cag.2006.07.003

Oliveira, D., Bezerra, B., Freitas, E., & Maciel, A. (2015). Adoption of Software Product Line to a Voice User Interface Environment. doi: http://doi.org/10.18293/SEKE2015-185

Martinez, J., Sottet, J. S., García, A., Ziadi, T., Bissyandé, T., Vanderdonckt, J., Klein. J., & Le Traon, Y. (2017). Variability Management and Assessment for User Interface Design. In: Sottet, JS., García Frey, A., Vanderdonckt, J. (eds) Human Centered Software Product Lines. Human–Computer Interaction Series. Springer, Cham. doi: https://doi.org/10.1007/978-3-319-60947-8_3

Markopoulos, P. (2021). A Compositional Model for the Formal Specification of User Interface Software Retrieved from https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.263807

Mazo. R. (2018). *Guía para la Adopción Industrial de Líneas de Productos de Software*. Editorial Eafit.

Motara, Y. M. (2020). String Diagrams for Modelling Functional Programming," 2020 2nd International Multidisciplinary Information Technology and Engineering Conference (IMITEC), Kimberley, South Africa, 2020, pp. 1-7, doi: https://doi.org/10.1109/IMITEC50163.2020.9334072

Motara, Y.M. (2021). High-Level Modelling for Typed Functional Programming. In: Zsók, V., Hughes, J. (eds) Trends in Functional Programming. TFP 2021. Lecture Notes in Computer Science, vol 12834. Springer, Cham. doi: https://doi.org/10.1007/978-3-030-83978-9_4

Pałka, M. (2007). Functional Graphical User Interfaces — An Implementation based on GTK. Retrieved from http://www.ens-lyon.fr/LIP/Pub/Rapports/DEA/DEA2007/Master2007-04.pdf

Peffers, K. Tuunanen, T., Rothenberger, M., Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. doi https://doi.org/10.2753/MIS0742-1222240302

Pleuss, A., Hauptmann, B., Dhungana, D., & Botterweck, G. (2012). User Interface Engineering for Software Product Lines: The Dilemma between Automation and Usability. doi: https://doi.org/10.1145/2305484.2305491

Pleuss, A., Hauptmann, B., Keunecke, M., & Botterweck, G. (2012). A case study on variability in user interfaces. doi: https://doi.org/10.1145/2362536.2362542

Pohl, K., Böckle. G., & Linden, F (2005). Software Product Line Engineering: Foundations, Principles and Techniques. Springer Publishing. doi: https://doi.org/10.1007/3-540-28901-1

Rabiser, R., Grünbacher, P., & Dhungana, D. (2017). Software Product Line Engineering. In Engineering of Software (pp. 297-314). Springer International Publishing. doi: https://doi.org/10.1007/978-3-319-52944-9_14

Rodríguez, Francy D., Acuña, Silvia T., and Juristo, Natalia (2015). Design and programming patterns for implementing usability functionalities in web applications. doi: https://doi.org/10.1016/j.jss.2015.04.023

Sboui, T., Ben Ayed, M., & Alimi, A. (2018). A UI-DSPL Approach for the Development of Context-Adaptable User Interfaces. IEEE Access, vol. 6, pp. 7066-7081, doi: https://doi.org/10.1109/ACCESS.2017.2782880

Schartum, S. & Borgersen, J. (2020). Composing Software Product Lines with Machine Learning Components. Retrieved form https://www.duo.uio.no/bitstream/handle/10852/79489/1/Master-Thesis.pdf

Schmidt, A., Mayer, S., & Buschek, D. (2021). Introduction to Intelligent User Interfaces. doi: https://doi.org/10.1145/3411763.3445021

Shankar, A., Louis, S., Dascalu, S., Hayes, L., & Houmanfar. R. (2007). User-Context for Adaptive User Interfaces. doi: https://doi.org/10.1145/1216295.1216357

Schweiker, K. Varadarajan, S., Spivak, D. I., Schultz, P., Wisnesky, R., & Marco, P. (2015). Operadic Analysis of Distributed Systems. Retrieved from http://www.dspivak.net/grants/Report_OADS.pdf

Smaragdakis, Y. (2020). Software Systems Compositionality. *In Proceedings of the 2020 ACM SIGPLAN International Conference on Software Language Engineering* (pp. 1-11). doi: https://doi.org/10.1145/3426425.3426935

Stuerzlinger, W., Chapuis, O., Phillips, D., & Roussel, N. (2006). User Interface Façades: Towards Fully Adaptable User Interfaces. doi: https://doi.org/10.1145/1166253.1166301

Weld, D., Anderson, C., Domingos, P., Etzioni, O., Gajos, K., Lau, T., Wolfman, S. (2003). Automatically personalizing user interfaces. *In: Proceedings of the International Joint Conference on Artificial Intelligence,* pp. 910–950, Acapulco, Mexico (2003).

Wigdor, D. (2010). Architecting Next-Generation User Interfaces. doi: https://doi.org/10.1145/1842993.1842997

Wąsowski, A., Berger, T. (2023). Software Product Lines. In: Domain-Specific Languages. Springer, Cham. doi: https://doi.org/10.1007/978-3-031-23669-3_11

Yigitbas, E., Josifovska, K., Jovanovikj, I., Kalinci, F., Anjorin, A. & Engels, G. (2019). Component-Based Development of Adaptive User Interfaces. doi: https://doi.org/10.1145/3319499.332820029