NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

JOÃO RAFAEL MIRA DE CARVALHO PACHECO

Master/BSc in Computer Science

COLLABORATION BETWEEN DEVELOPERS AND DESIGNERS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

February, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

COLLABORATION BETWEEN DEVELOPERS AND DESIGNERS

JOÃO RAFAEL MIRA DE CARVALHO PACHECO

Master/BSc in Computer Science

Adviser: Miguel Goulão
Associate Professor, NOVA University Lisbon

Co-adviser: Stoyan Garbatov
Research & Development Engineer, OutSystems

Examination Committee

Chair: João Miguel da Costa Magalhães
Associate Professor, NOVA University Lisbon

Rapporteur: Jácome Cunha
Associate Professor, University of Porto

Adviser: Miguel Goulão
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 2022

Collaboration Between Developers and Designers

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.5) [52].

Acknowledgements

Firstly I would like to thank my thesis adviser Miguel Goulão for giving me valuable advice and for picking me, and giving me, the opportunity to work alongside him and OutSystems. I joined FCT (now NOVA) only for my master's and due to this plus the pandemic, I never had the opportunity to meet many of the professors at the University including professor Goulão. I can safely say it was great "shot in the dark", as the professor's down-to-earth approachable personality made it a joy to work with.

Secondly I would like to thank my OutSystems' co-adviser Stoyan Garbatov. Your guidance through the process and insight within OutSystems were fundamental. The schedules we defined during our weekly meetings made sure I never slouched and your words kept me motivated throughout the whole work. You forced me into situations that I was not comfortable with such as (many, many) presentations and interviews but that ultimately made me grow as a person and I can safely say I am much more comfortable now. It was a joy to meet and work with you.

Finally I would like to thank my friends and family. This year has been rough on all of us but your presence and kindness made this lockdown/stay-at-home 24/7 not only bearable but some times very enjoyable. Special mention to the ones that kept me company during many afternoons online and my parents for putting up with me.

"Science isn't about why, it's about why not!" (Cave Johnson)

Abstract

Customer-facing applications are essential for businesses. Therefore, a good user experience is fundamental for their success in the market. Companies nowadays employ highly specialized people in front-end development and User Experience (UX) & User Interface (UI) design to achieve this goal. Their collaboration is critical, and raises some efficiency challenges in the software industry. This work focuses and is applied on OutSystems, a low-code platform that inherits these challenges.

While there are some code-generation plugins for popular design tools, these do not generate code for low-code platforms. Therefore, the transformation process from design to development is done 100% manually, which is highly inefficient. Our goal is to accelerate this transformation process from a design model to a development model to mitigate this inefficiency.

To do so, we developed an approach using model transformation techniques that automates part of the process. Namely, it automates the generation of application pages/screens by composing the screen mockups in a design technology (such as Figma or Sketch) with a library of reusable UI components to instantiate the design in a front-end technology (such as OutSystems).

Our approach was validated by a professional team of front-end developers from an established enterprise-grade low-code platform who applied and evaluated this work on some of their past real projects. Preliminary results show an overall acceptance of the developed tool with a possible increase of 150% to 400% in the value that they can deliver without investing more effort than they already do today. This mitigates a bottleneck faced by development teams today. To increase the value, they could offer to customers (e.g., by producing more application screens in the same period), they would need to recruit new collaborators whose skill set is high on demand. This work delivers major efficiency improvements and lessens the severe lack of qualified professionals, by allowing existing ones to produce more without investing further effort.

Keywords: Design To Code, OutSystems, Low-Code Platforms, Front-end Development, Automation, Generation

Resumo

As aplicações são algo essencial para as empresas. Uma boa experiência de utilizador é fundamental para o sucesso destas aplicações no mercado. Hoje em dia, para alcançar este objetivo, as empresas empregam pessoas altamente especilaizadas em desenvolvimento Front-End e de UX (User Experience) & UI (User Interface) design. A colaboração destas equipas é crucial e de momento apresenta desafios de eficiência na indústria do software. Este trabalho foca-se na OutSystems, uma plataforma low-code, que tem subjacente estas ineficiências que estão presentes em toda a industria.

Embora atualmente existam alguns plugins de geração de código para as ferramentas de design populares, estes não geram código para plataformas low-code. Portanto, o processo de transformação de design para desenvolvimento é um processo 100% manual, o que resulta em perdas de eficiência que serão refletidas no valor final entregue aos clientes. O nosso objetivo é acelarar este processo de conversão de um modelo de design para um modelo de desenvolvimento Front-End para mitigar esta ineficiência.

Para tal, desenvolvemos uma abordagem utilizando técnicas de transformação de modelos que automatizam parte do processo. Nomeadamente, este automatiza a geração de páginas/ecrãs de aplicações através da composição de mockups de ecrãs numa tecnologia de design (como o Sketch) com uma biblioteca de componentes de UI reutilizáveis para instanciar o design numa tecnologia Front-End (como a OutSystems).

A nossa abordagem foi validada por uma equipa profissional de desenvolvimento Front-End de uma plataforma low-code de nível empresarial que aplicaram e avaliaram o trabalho em projetos passados reais da equipa. Os resultados preliminares mostram uma aceitação global da ferramenta desenvolvida, com um possível aumento entre 150% a 400% no valor que conseguem oferecer.

Isto permite mitigar um ponto de fricção que as equipas de desenvolvimento encontram de momento. Para aumentar o valor que a equipa consegue entregar aos clientes (por exemplo, através da produção de mais ecrãs no mesmo período de tempo), estes necessitariam de empregar novos colaboradores cujas habilidades têm elevada procura. O nosso trabalho oferece uma alternativa mais económica para o aumento da eficiência e ao mesmo tempo diminui o impacto da escassez de profissionais qualificados, ao permitir

que os já existentes consigam produzir mais sem investimento adicional da sua parte.

Palavras-chave: Design To Code, OutSystems, Low-Code, Desenvolvimento de Front-end, Automação, Geração

Contents

Annexes

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Description

The number of people using digital devices and digital applications has increased considerably. According to a recent report posted by Statista, a German statistics portal, the number of people who accessed the internet in 2020 was 4.66 billion[42]. Smartphone usage from 2012 to 2019 has tripled to 3.2 billion with a prediction of 3.8 by the end of 2021[67]. With a worldwide population estimate of 7.74 billion people[21], this represents 41% of the world population who has a smartphone. End-users expectations when interacting with a digital application, such as its usability and performance are also continuously growing [19]. Customer Experience (CX) is a significant factor in the market and can set a product apart from the competition [69, 90, 22]. The concept of Customer Experience includes many different aspects, one of them being Design. Gartner has reported that more than two-thirds of companies now "compete primarily on the basis of customer experience" [85].

Norman first coined the term User Experience (UX) in 1990[66], which refers to the users' experience when interacting with a product. In his book, he also proposed a shift towards a more user-centered approach to focus more on the users' needs when developing a product. Since then, UX has been an integral part of designing products, especially when designing applications, as it is a critical aspect in their reception.

Collaboration between people of diverse expertise and professional qualifications has been a general necessity for every industry [44, 45] and the software development industry is no exception. A research by Faraj and Sproull [25] has shown that the importance of well coordinating different expertises can overtake, in relevance, the existence of these expertises in the first place. A product's quality and value often reflect the teamwork between a combination of different roles. In 2016, Lindsjørn et al, investigated the relation between teamwork quality and the quality of the product, focusing more on teams that employed an agile development strategy. This research concluded that the quality of teamwork plays a significant role, not only in terms of optimizing the team's performance, but also in increasing the quality of the product [50].

Today, developing software requires work and input by collaborators with distinct profiles and skills. A particularly relevant collaboration is that between the UX/UI design and front-end development practices. Designers design the entire system's expected behavior, look, and feel, while developers turn the designs into reality through a front-end technology. One role cannot create the whole system at an enterprise-grade level. UX and UI designers lack the technological knowledge developers possess, and, as stated by Norman, regarded as the "father" of UX, developers/engineers are "trained to think logically" and design interfaces for "people the way they would like them to be and not for how they are,"[65] which often results in lackluster user experiences. This means the collaboration between these areas is key to developing a successful product.

The translation from design to web or mobile technology is often done by Front-End developers based on the designers' UX and UI designs, more frequently after the UI design is finished. Each team uses different tools dedicated to their fields as these tools present many advantages for these professionals to exercise their work. To the best of our knowledge, no current unified tool provides these kinds of advantages. This results in developers having to translate from the design format into a front-end technology manually by looking at each component or inspecting them with "Engineering handoff tools" [37]. These methods are very time-consuming, no matter the developer's expertise. This work focuses on the OutSystems ecosystem, a low-code platform that shares these inefficiencies in the collaboration between UX & UI designers and front-end developers. An applied use case will be targeted at the Customer Success Department of OutSystems, where currently developers compose these pages manually. These developers do this by manually dragging and dropping the UI elements into the pages using their IDE Service Studio while inspecting the designs' properties to make both correspond as best as possible. Any sort of mitigation of this inefficiency would increase the value produced by these teams and any other with similar issues.

1.2 Motivations

There are inefficiencies with the collaboration between the design and the development practices. One of these inefficiencies lies in the transformation of pages from design to a web or mobile technology as it is often done manually by front-end developers. With our work, we aim to mitigate this inefficiency. Automating this process will allow a better return of investment, even if just partially automating it, as developers can dedicate their time and focus to issues that better deserve their expertise. This will allow teams to create more value for their customers without extra investment on their part.

Another motivator lies in the current and continuously growing relevancy of UX and UI because of the people's expectations being higher than ever. Companies have noticed this continuous growth and have prioritized providing their customers a good user experience as it is a determinant factor in the market. Alongside the importance of the UX and UI factors in the market, another big aspect is a company's ability to react to

the continuously evolving market. Often referred to as "First Mover Advantage"[48], companies that are among the first to react to changes in the market gain a significant competitive advantage over their competition as businesses that come after. However, the difficulty of hiring skilled professionals in Design and Front-End technologies makes these a challenging point for companies. This current challenge makes the collaboration between designers and developers even more important.

One of the more recent changes in the market is the surfacing of low-code platforms. These platforms allow developers to be more efficient by providing more value with the same or lower efforts on their part than the more traditional methods. They allow people with different backgrounds that are not necessarily computer science to solve business relevant problems (through applications they create), since they require lower programming knowledge. These factors result in the current popularity these platforms present today and will keep on gaining in the future [92, 98].

1.3 Objectives and Expected Results

This work proposes an approach, and consequently, a tool that implements it, that automates the creation of pages in a front-end technology through the composition of common reusable UI components. These are common components that are, as the name suggests, reused throughout the whole application such as buttons, dropdown menus and accordions. Composing custom patterns, which are patterns that are not included in off-the-shelf UI component libraries, is out of the scope of this work. Its principal focus is on providing a completely structured page by composing design artifacts with the provided UI components.

In the case of OutSystems, this tool is going to be applied to the creation of Sample Pages, which are fully structured and functional application pages in OutSystems (more on this in the next chapter). It is expected that in the same time that the Front-End experts have right now to create their set of Sample Pages (later more elaborated at section 2.1.6.2), they will be able to create more and, in doing so, will deliver further value without extra investment. It was also expected that this tool's effectiveness would depend on the projects' complexity, as more complex works are bound to have more custom patterns that we are not accounting for. To confirm this statement and test its overall performance, the created tool was applied to several past real OutSystems projects with different degrees of complexity, which was then evaluated by a professional team of OutSystems Front-End experts. Ideally, by the end of the process, a Sample Page would be completely structured for every screen design. The degree of automation will be conditioned by the volume of custom patterns and how many components we can identify correctly.

Looking at a broader picture and outside of OutSystems scope, it is expected that this work's generic and technology-independent architecture can serve as a stepping stone in developing automated services that provide the ability to compose structured pages

by the given design artifacts and UI components in a web or mobile technology different from the ones tested here.

1.4 Document Structure

This chapter provided an introduction to the work. The remainder of the document is structured as follows:

- **Chapter 2 - Context and Background:** This chapter aims to provide the relevant context in the OutSystems Customer Success current workflow and show where the work will be applied.
- **Chapter 3 - Related Work:** Here, we present relevant work that is related to this thesis. We discuss their main contributions and reason on how they do not apply or differ in our context.
- **Chapter 4 - Approach:** This chapter presents high-level solutions along with their pros and cons, more in-depth views of the solution and challenges that are deep rooted in approaches like mine.
- **Chapter 5 - Results:** This chapter presents the results gathered from our internal testing where we calculate the Precision and Recall values of 6 real past OutSystems projects and the feedback gathered from the Front-End Expert team over the same 6 projects.
- **Chapter 6 - Discussion:** This chapter presents an analysis of the results gathered from the previous chapter as well as some points regarding the whole project.
- **Chapter 7 - Conclusion:** This chapter presents a final conclusion to this thesis alongside some future work proposals.

2

Background And Context

2.1 OutSystems

To gather a deeper understanding of how OutSystems works, specifically their Customer Success department, their established practices, and current workflow, we conducted interviews with experts in their respective fields to complement our literature research. To complement the knowledge about their platform and IDE Service Studio, OutSystems provided tutorials, that gave an high-level overlook of their architecture [77].

2.1.1 Company Overview

OutSystems is a software company that focuses on improving the productivity and efficiency [70, 75, 76] of building digital applications while at the same time decreasing the minimal requirements on technical background necessary for doing that. To this end, OutSystems built a low-code development platform. The low-code approach allows developers to create and evolve digital systems without worrying much about low-level implementation details, resulting in increased agility and speed to achieve business goals [72]. This approach allows people from different backgrounds, from a wide range of computer systems and IT knowledge, to develop their very own enterprise-grade applications. OutSystems has various teams and departments. This work will focus on their Customer Success department.

2.1.2 OutSystems Applications

The OutSystems platform allows for multiple types of applications to be developed.

Traditional Web Applications are applications that focus their workload on the server-side (rendering and logic operations) and synchronous data fetching.

Reactive Web Applications are applications that share logic between server-side and client-side, support asynchronous data fetching and do the rendering client-side. These applications follow more closely today's technological frameworks such as React [36], Vue [100] and Svelte [96] by allowing the creation of reactive components that adapt to different screens.

While the Traditional Web Applications and Reactive Web Applications are, as the name implies, based on Web technologies, OutSystems also allows the creation of **Mobile Applications** which compile to an Android or iOS application.

2.1.3 Customer Success Department

Customer Success is a department in OutSystems responsible, among other things, for creating and delivering projects to their customers. This division has various services they can provide. For this work, I will pay special attention to a kind of service this division provides called Accelerator Packs, as these provide the customers with a Live Style Guide.

2.1.4 Live Style Guide

Live Style Guides [74], or LSG, are OutSystems' libraries of reusable UI components, where all the guidelines regarding each project's appearance and behavior are organized. These are composed of all the commonly reusable UI elements (such as menus and buttons and can be extended with custom patterns) and Sample Pages for a specific application. An example of a subset of UI patterns in an LSG can be seen in Fig.2.1. The goal of LSGs is to foster consistency, improve the developers' overall efficiency and productivity when constructing screens for their application while lowering the required skill set to do so, and lower the page's components' maintenance cost.

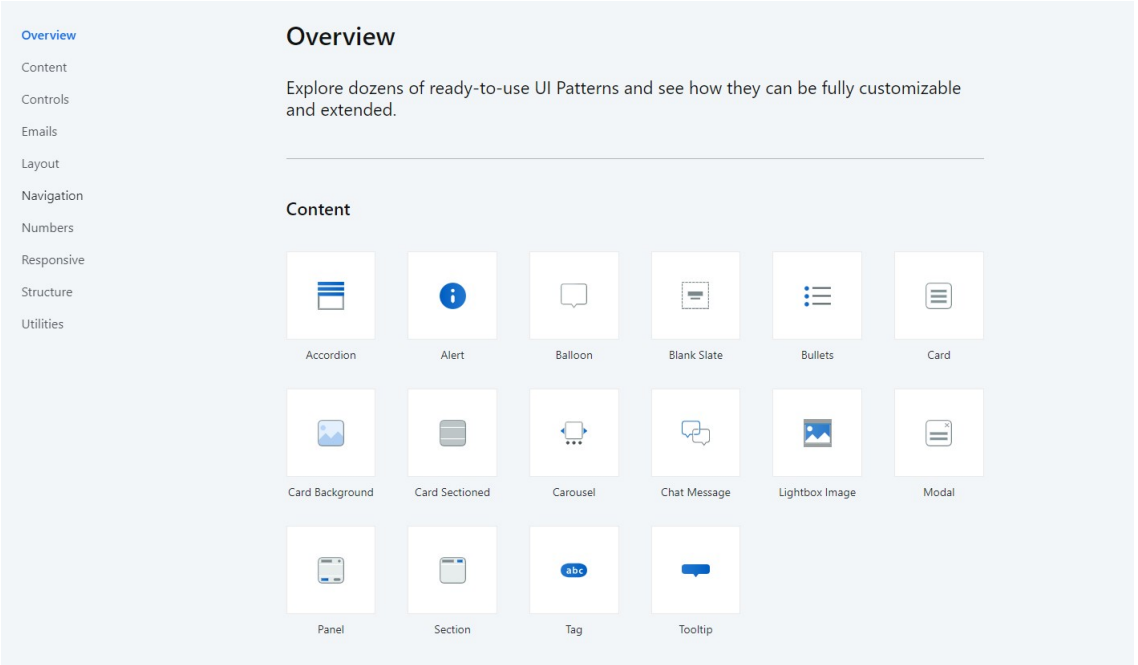


Figure 2.1: UI patterns from OutSystems' Live Style Guide [79]

Live Style Guides accomplish these goals by implementing a variation of the Atomic Design methodology[28] proposed by Frost (Fig.2.2).

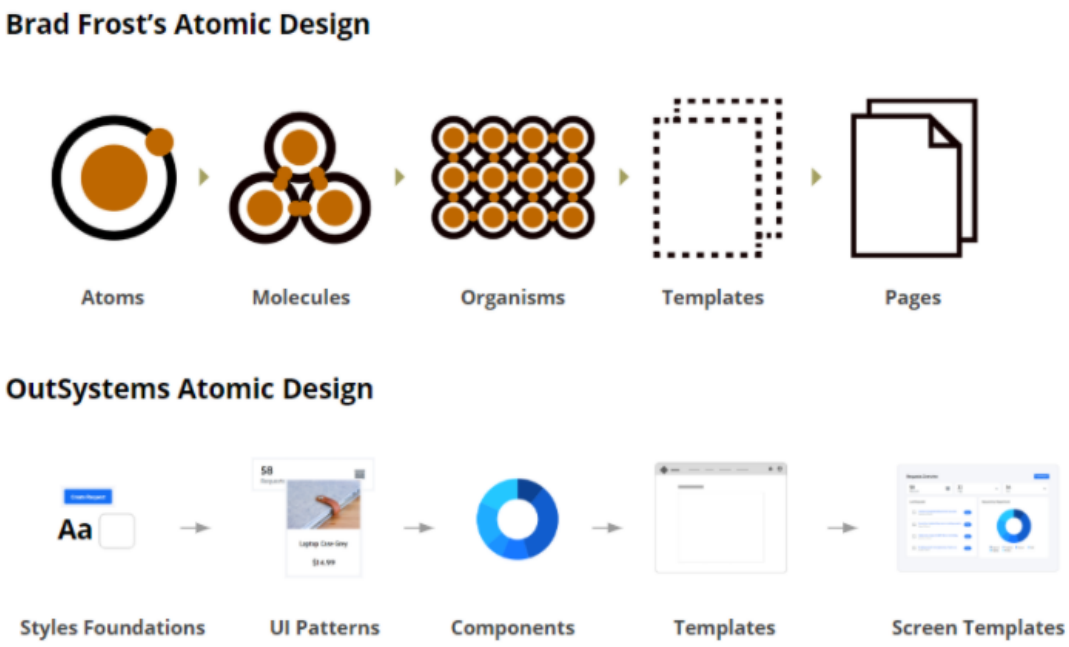


Figure 2.2: Frost's Atomic Design methodology applied to OutSystems[77]

Frost proposed a methodology to craft interface design systems that took Chemistry as its major influence. He picked up the notions of Atom (the basis of all matter), Molecule (groups of Atoms), and Organism (groups of Molecules) and translated them into the world of User Interfaces, then added two more concepts: Templates and Pages. The Atoms of User Interface, just like in Chemistry, are the basic building blocks that form them. These can be buttons, inputs, labels, and other essential UI elements. These elements cannot be broken down without losing their functionality, but they do not possess a purpose just by themselves. The Molecules are simple groups of UI elements (Atoms) that function together as a unit. Frost's example is that of a search form, where it is composed of three different UI Atoms: a label, an input field, and a search button. Now this group of UI components has gained a purpose. The label describes what the user needs to put in the input field, and the button submits the form. Organisms are more complex groups of UI components that can be Atoms, Molecules, or other Organisms. An example of these can be the headers and footers we see on the Web. These groups form a standalone section of an interface. Templates serve as a model to be instanced by different Pages. These are Templates for Pages that contain their entire structure through the use of organisms and molecules, presenting no content such as images or text. We can draw a comparison of these to low-fidelity mockups which are, in general, composed only of functional components in their "bare" natural shape (no styling) and placeholders. Pages present the final interface that users will see. These Pages are instances of the Templates with actual representative content in place. This passage from Templates to Pages is like what the Designers do when passing from low-fidelity mockups to high-fidelity.

Figure 2.3 illustrates this methodology with an example.

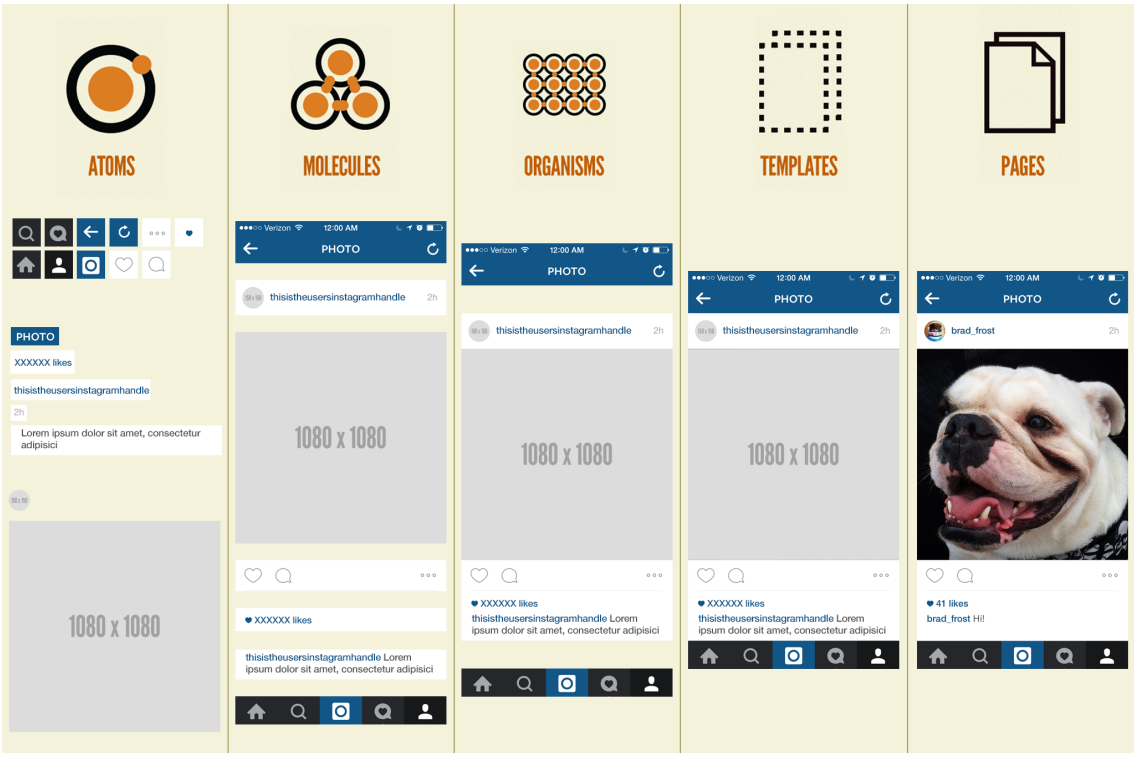


Figure 2.3: Representation of Frost's Atomic Design methodology [29]

Having access to a library with already entirely constructed reusable UI components lowers the required skillset from the developing team and, at the same time, allows these developers to achieve higher efficiency and productivity since they rarely have to start from scratch. The ability to reuse these UI components throughout the application promotes a higher level of consistency and low maintenance costs since the elements in the application's pages are instances of the source reusable UI component in the LSG. One change applied to the source will reflect through the instances.

Sample Pages are fully functional pages created by the Customer Success team for the customer. For each particular customer's business and business model, these pages are created and personalized. These pages are mostly structured by the reusable UI elements in the LSG and then are populated with mock data to allow the customer to test the system's interactions while serving as a stepping stone for the customer to build on.

2.1.5 Accelerator Packs

Accelerator Packs are services provided by OutSystems to their customers. The goal of these services is to help new customers build their first app. Figure 2.4 provides an overview of the whole process.

These services provide perks to the customers, such as supplying them with the designs, workshops to better understand the OutSystems platform and architecture so they are able to take full advantage of them, and create the LSG.

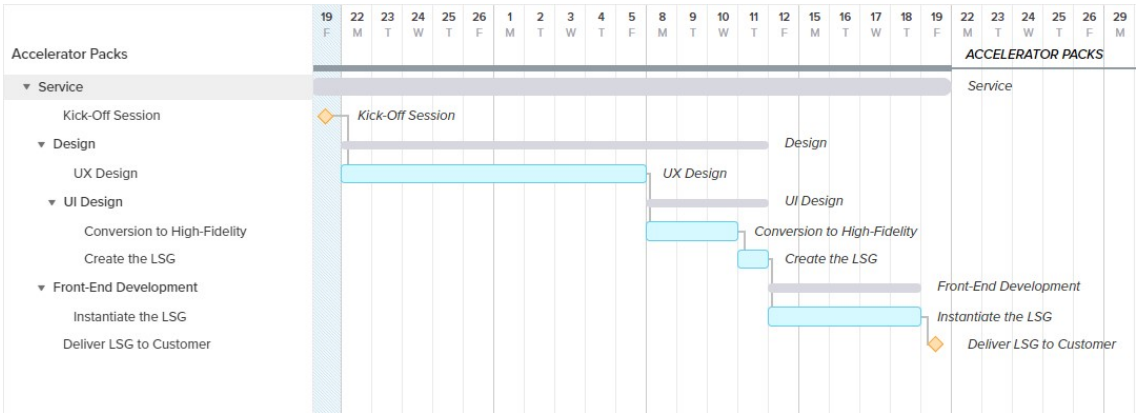


Figure 2.4: Quick overview of the Accelerator Pack service

The Customer Success department provides three different kinds of Accelerator Packs: Basic, Standard and User Experience. These three packs differ in what they provide for the customer. These differences are not relevant for this work since all three of these Accelerator Packs provides the customer with an **LSG** at the end of the service.

These Accelerator Pack services result from collaborative work between designers and developers and are divided into 2 phases: The design phase and the development phase. These are single direction processes, which means once the design process is over and the development begins, in general, there are no more changes to the design. There are exceptions when a Developer has to discuss some design aspects with the designer, but these occurrences are rare.

The Accelerator Pack service starts with a meeting between the customer's representatives and OutSystems design and front-end teams. This meeting's goal is for the customer to hand over any of its resources and brand guidelines, such as their logo, font, colors, among others, to the design team. The customer and developer team discuss and agree upon which Sample Pages the developer team will deliver by the end of the service.

After the meeting has ended, the design process begins. This process usually takes two weeks and is divided into two significant steps: the UX and the UI Design phases. The UX Design phase takes usually ten days. In this phase, the UX designer creates the low-fidelity mockups and conducts usability testing as well.

Once the UX phase finishes, the UI designer picks up the created mockups and proceeds to transform them into high-Fidelity. The designer in charge has four days to create these high-Fidelity mockups and the corresponding LSG.

Front-End developers pick up these high-Fidelity artifacts, and they usually have five days to instantiate the Live Style Guide. They first begin by instantiating the reusable UI elements in the OutSystems language. Once they have composed the elements, they begin to construct the Sample Pages with the same elements created in the step before.

By the end of the service, there are 2 OutSystems applications delivered to the client as can be seen in Figure 2.5: the Theme application and the LSG application [16]. The Theme application contains all of the project's styles and custom patterns. Since this

application is solely focused on the *theming* of the final application, it does not affect this work as styling and custom patterns are out of scope of this thesis.

The Live Style Guide application however, contains vital points for this work, namely, the LSG module. This module includes a live demo of the client adapted OutSystems UI widgets and custom patterns, as well as the created Sample Pages. This module, usually contains 2 to 4 Sample Pages (2 for Web Applications, 4 for Mobile) of medium complexity. As said in 1.3, it is not expected that this solution will help create every screen in the application. However, as stated by the interviewed developers, even if they can deliver one more Sample Page, it is a significant increase in the value they are providing to their customers.

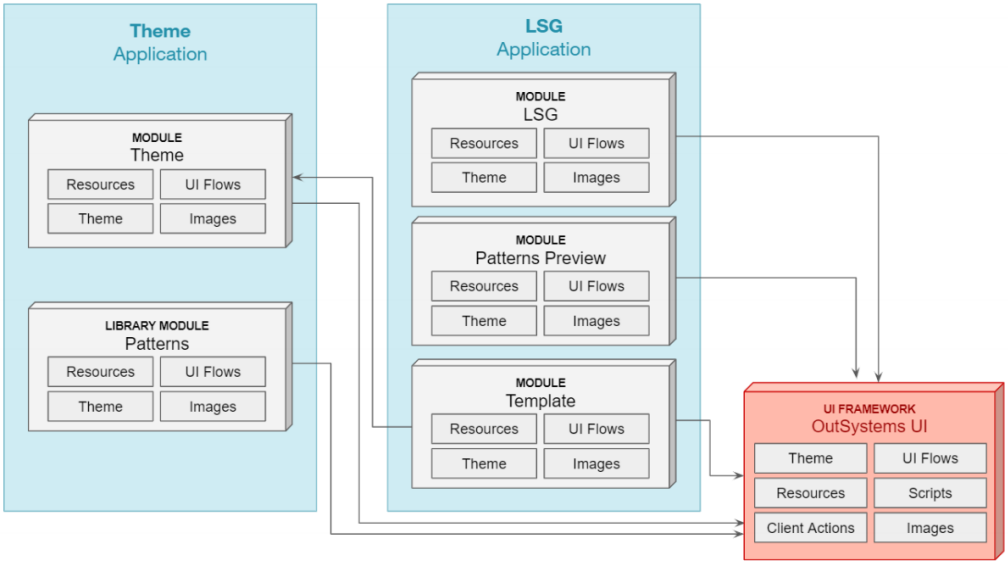


Figure 2.5: "LSG OutSystems Applications Organization"[16]

2.1.6 Current Workflow of the Design and Development Stages

To understand the current collaboration process, I interviewed experts in both Design and Front-End practices at OutSystems. These 1 hour interviews were semi-structured, as although I set out with prepared questions I was not restricted to these, which allowed us to discover new topics that were missed or were completely unbeknownst to us. The full list of questions I had structured can be found at Annex I.

2.1.6.1 Designers

The following information was gathered by interviewing three designers at OutSystems that have experience in the Customer Success department. The questions made in the first interview were more generic as the goal was to gather more information on their workflow, while the following two helped solidify any assumptions and identify possible risks.

The interview started with a few professional background questions, so we could better know our interviewees and properly frame their experience. This was followed by more technical questions about their practices in the Customer Experience department, including (the full interview questions are presented in section I.1 of the Annex):

- Do you receive any sort of mockups from clients? If so, in what format do they come in and what do you do with them.
- Do you create the UI components from a reference point such as a template and if so, how updated is it.
- How many Sample Pages do you design?
- What are the criteria used to pick the Sample Pages to design?

Designers define the look and feel of the entire system. They use Sketch[13] as their design tool, and it is regularly one designer per phase throughout the entire design process. The process starts with the creation of low-fidelity mockups (Fig 2.6). The creation of these mockups is always done by utilizing elements defined in a template called the OutSystems UI LSG Template, and designers tend to create a mockup for every screen of the system. This template possesses all of the reusable UI components in their "bare" natural state (no styling).

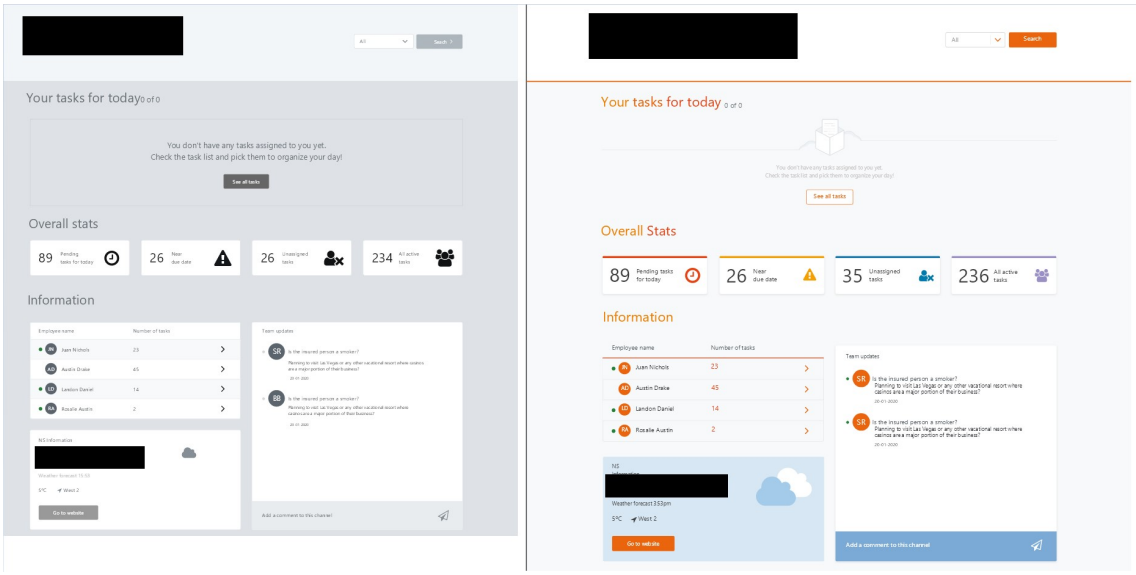


Figure 2.6: Side by side view of a low-fidelity(left) mockup with an high-fidelity(right) mockup of the same project

After the ten days have passed, the designer in charge of the UX phase delivers the created mockups to the designer in charge of creating the UI. This designer has three days dedicated to the UI design (Fig 2.6), and the last day is solely dedicated to creating the LSG. This new LSG has all of reusable UI components in their final iteration (i,e.

colors/dimensions/any sort of styling property is set) and screen mockups. This model follows an established practice that dictates that the pages should be made before the LSG. Nevertheless, through the conducted interviews, I concluded that some designers prefer to do both jobs(UI design and LSG) simultaneously to improve consistency and productivity. These designers bootstrap their LSG while working concurrently on the pages as it provides them with the perks of working with an atomic design methodology.

The established method presents a challenge for this work since the elements in the pages are decoupled from the ones in the LSG. As such there is no guarantee a relation between them can be established. This coupling is needed as I am trying to automate a process, and I need to be able to correctly identify these elements so we can map them to the OutSystems language.

After completing the UI phase, the designer uploads the LSG to inVision, a tool that has a dedicated feature to help the handoff phase between design and development.

2.1.6.2 Developers

To get a deeper understanding of the front-end developers workflow at OutSystems, I conducted interviews with two front-end experts at the Customer Success department. Just like with the designers, the first developer interview was more for us to get an insight of the whole workflow while the second was more focused on clarifying some details. Also like the designers, the interviews started with a few professional background questions, so we could better know our interviewees followed by more technical questions about their practices in the department. These were as follows:(the full interview questions are presented in section I.2 of the Annex):

- What do you receive from the design team?
- How do you collaborate with the design team?
- Do you create everything manually just by looking at the designs?
- What are your biggest challenges?

The front-end development process kicks off when the design step finishes. Their work revolves around instantiating the design artifacts into a functioning system in OutSystems. As stated before, their **first** phase is instantiating the reusable UI elements in OutSystems since these will be the stepping stones of the system and will facilitate the next step. This work is done manually by observing and inspecting the design artifact elements (that have been uploaded to inVision) or automated utilizing a tool created by Bexiga [16, 17]. When these elements are completed, they begin the **second** phase: constructing the Sample Pages.

This whole phase is separated into **three** major steps:

1. Constructing the page's structure by dragging and dropping the components created in the previous step.

2. Assigning any necessary parameters such as variables or auxiliary structures so they function correctly.

3. Making the pages more realistic by creating logic that bootstraps sample data.

The work presented in this dissertation will address the first activity, while also providing some validation for the second. The third activity is completely beyond the scope of this dissertation although it presents a good future work opportunity.

The front-end developers use **Service Studio**, the official OutSystems IDE to exert their work. At this point we have the complete development toolchain and it can be seen in Figure 2.7.

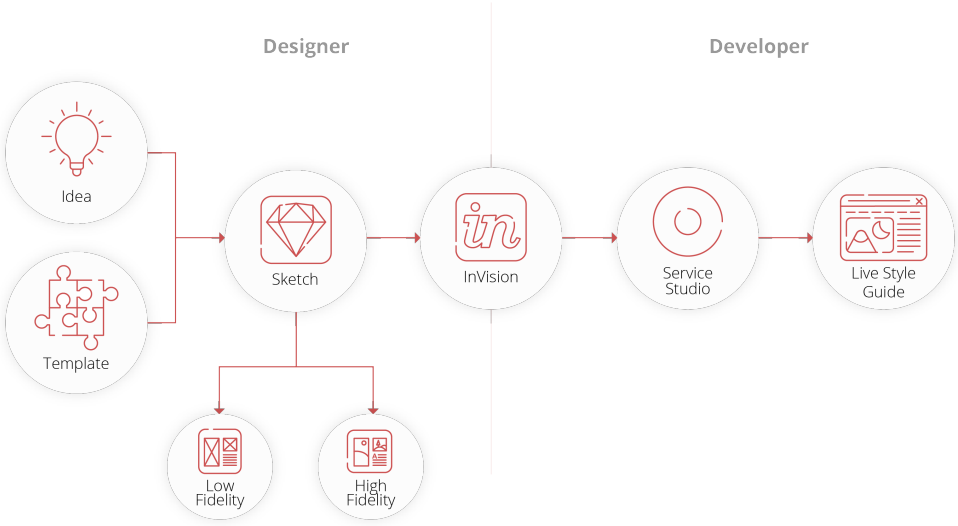


Figure 2.7: Development Tool Chain

2.2 Design Tools

There are various design tools in the market, each with different purposes to help designers create their works. These can be categorized as image editing tools, UX and UI dedicated tools, or collaboration tools.

Adobe Photoshop[2], GIMP[31], and Paint.NET[51] are examples of image editing tools. These tools' goals are towards image manipulation and creation through different mechanisms they provide. This set of tools does not apply in my context as they do not support the design of interfaces, nor are dedicated tools for collaboration between groups.

Sketch[13] and Figma[27] are two of the most well-known tools[83] in this category. These tools are focused on supporting the User Experience and User Interfaces' design process and provide many advantages to designers to work more efficiently, such as providing them with pre-made assets or features like Sketch's Symbols which are applications of the Atomic Design method. This category contains applications that can be more focused on the side of the UX, such as Balsamiq[14] and Pidoco[86], or more focused on the UI like Figma and Adobe XD[3].

Collaboration tools, or "Engineering handoff tools,"[37] specialize in assisting the hand-off between the designers and developers through features such as file-sharing or code inspection. These tools can be solely dedicated to this purpose, like Zeplin[101], Avocode[6], and Sympli[97]. They function by importing designs from other design tools. They can also be incorporated into a UX & UI dedicated design tool, which is the case of Figma[68] and InVision[38].

At OutSystems, in the Customer Success department, the designers use Sketch as their design tool of choice. When they finish the designs, they use InVision as their choice of a collaboration tool to send the designs over to the development team. By entering InVision's "Inspect Mode" (Fig. 2.8), developers can click on any component they desire and InVision will present them a CSS snippet with the component's properties. Not only that, but since the designs are imported from Sketch, it will mention if the component corresponds to a Sketch Symbol and present its name (although through my research on past projects the name did not always correspond properly).

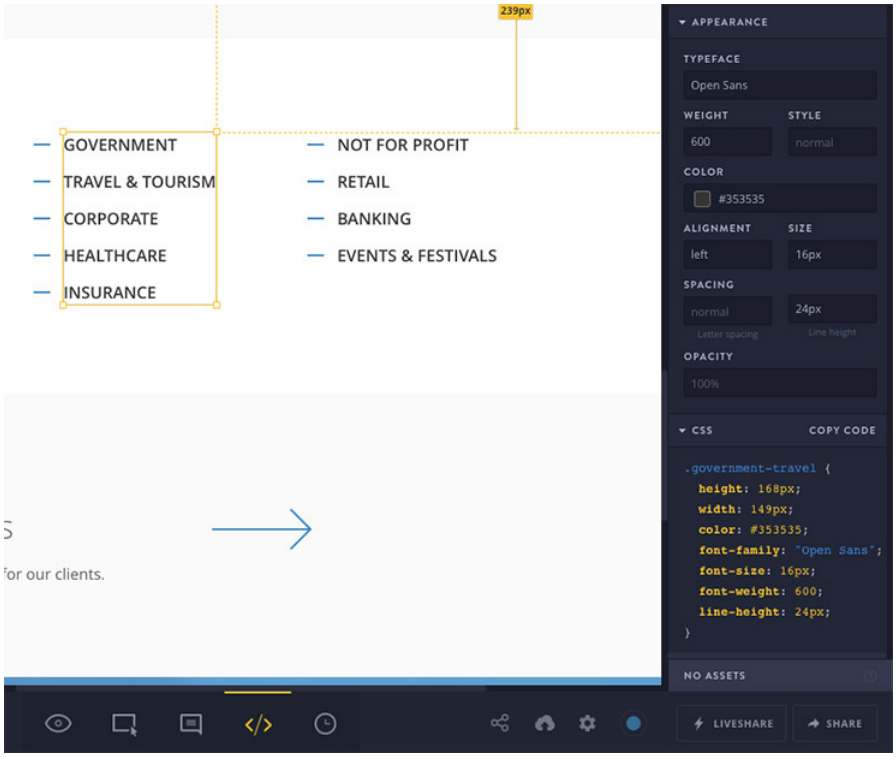


Figure 2.8: InVision Inspect example

2.3 Model Driven Engineering

Model-Driven Engineering is a software development methodology that enables development to be done at higher abstraction levels [5]. This methodology boasts many advantages in gains of productivity, maintainability, and portability [35], although many of these advantages are currently being misused in industries because of a lack of knowledge on applying an MDE methodology or even what MDE is [35]. MDE uses models to describe systems. These models follow the three main criteria proposed by Stachowiak: Mapping (*there is an original object that is mapped to the object* [54]), Reduction(the model must represent a set of properties from the original but not all), and Pragmatism (the model can replace the original for some purpose making it useful) [46, 54]. To *define the structure of a modeling language that models use* [93], we turn to meta-models. This language is referred to as the **modeling language** and can be defined as domain specific [23, 43, 55] or general-purpose [93] depending on its scope. Figure 2.9 illustrates the relationships between these elements.

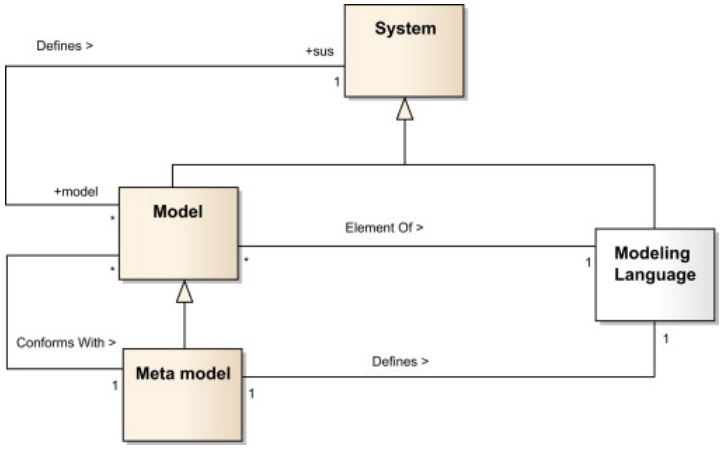


Figure 2.9: The metamodel definition: relationships between metamodel and model [93]

2.3.1 Model Transformation

Model Transformation is the process of automatically converting a source model into one or more target models following an established set of transformation rules [59].

A possible classification of model transformations is dependent on the nature of the source and the target model. When these coincide, we are in an **endogenous** transformation or *rephrasing*. Some examples are Optimization, Refactoring, Simplification and Component Adaptation [46]. If the models are from different natures, we are presented with an **exogenous** transformation or *translation*. These can be Synthesis, Reverse Engineering or Migration transformations [46]. My work's core is a model transformation process where we will transform our source model Sketch into our target model, the OutSystems language.

2.3.2 Models in my Work

This work has 2 models as it has been mentioned throughout the thesis: the Sketch model and the OutSystems model. These models are the basis of this work, as we are trying to transform our source Sketch model into the OutSystems target model, and so, understanding them and their composition is key.

2.3.2.1 Sketch Model

Sketch is a digital design toolkit developed by Sketch. B.V[13]. Its primary focus is on producing low and high-fidelity representations for user interfaces. Internally, Sketch is composed of "Layers." These Layers are the *building blocks for creating designs in Sketch* [10]. Some of the layer types that are important for this work are the following:

- **Group:** These are composed of multiple layers. There are specialized kinds of this layer type such as:
  - **Page:** These are groups that represent a canvas of the document. A simplified way of visualizing these pages would be referring to them as categories, i.e., a Page called UI would contain every UI design.
  - **Artboard:** These contain the designs of what would be the actual screens of the system.
  - **Symbol Master:** These are group layers that are frequently used around the whole document. Any change applied to them is propagated in their instances. [12]
  - **Symbol Instance:** These are instances of the Symbol Master and inherit their properties. However, these can alter some properties independent of the master.[12]
- **Text:** Layer type that contains the text.
- **Shape:** The most common type of Layers in Sketch. They represent pre-made shapes that are pre-built or created by the designer[11].

Layers have common properties that characterize their elements independently of their class, and others more specific such as the text properties in a layer with Class Text. The diagrams 2.10 and 2.11 showcase this model.

There are common properties that are shared between every layer. The most important are:

- **Name:** Represents the name given by the designer.
- **Class:** Represents the kind of the layer (mentioned above).

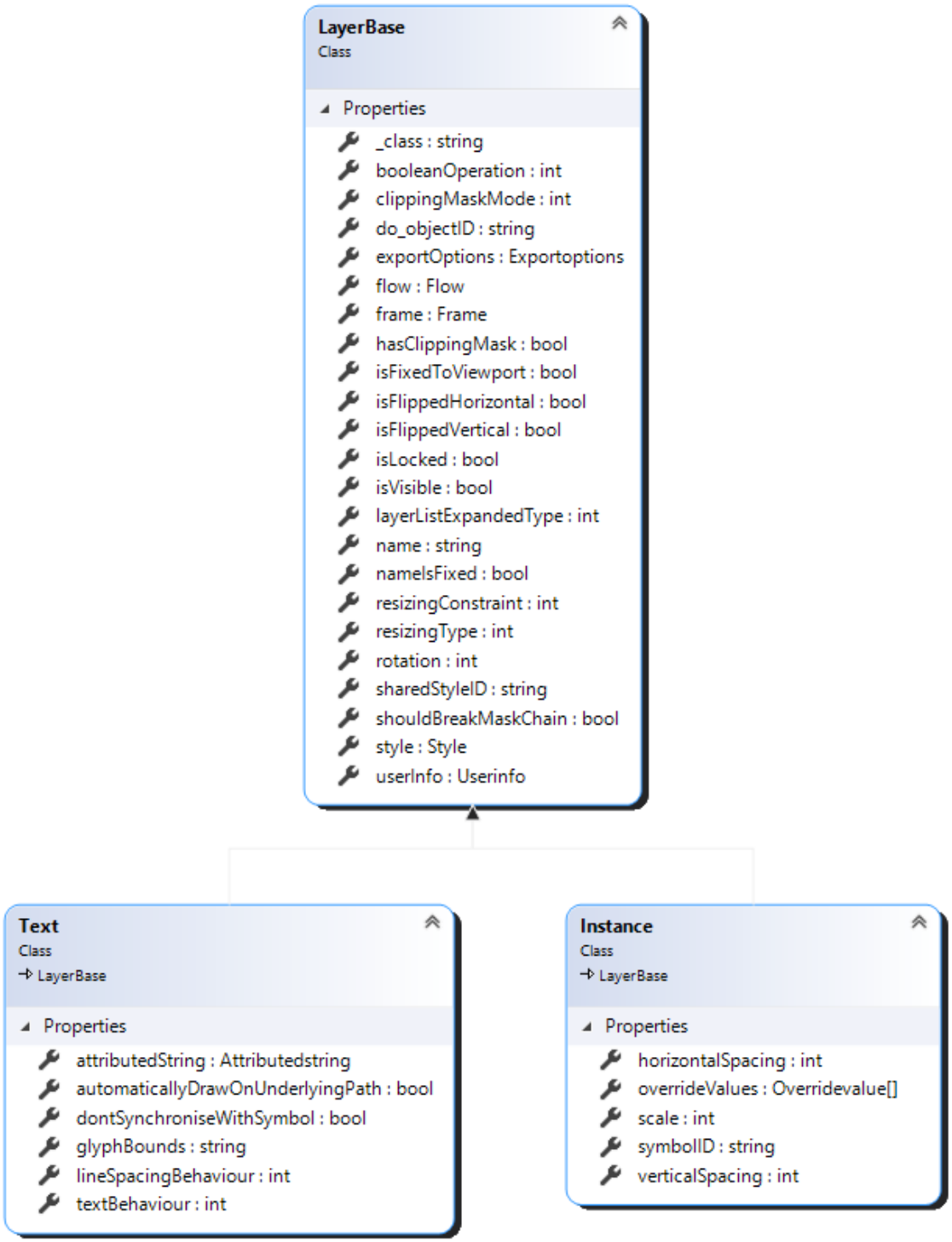


Figure 2.10: UML class diagram showcasing the Layer class and derived classes

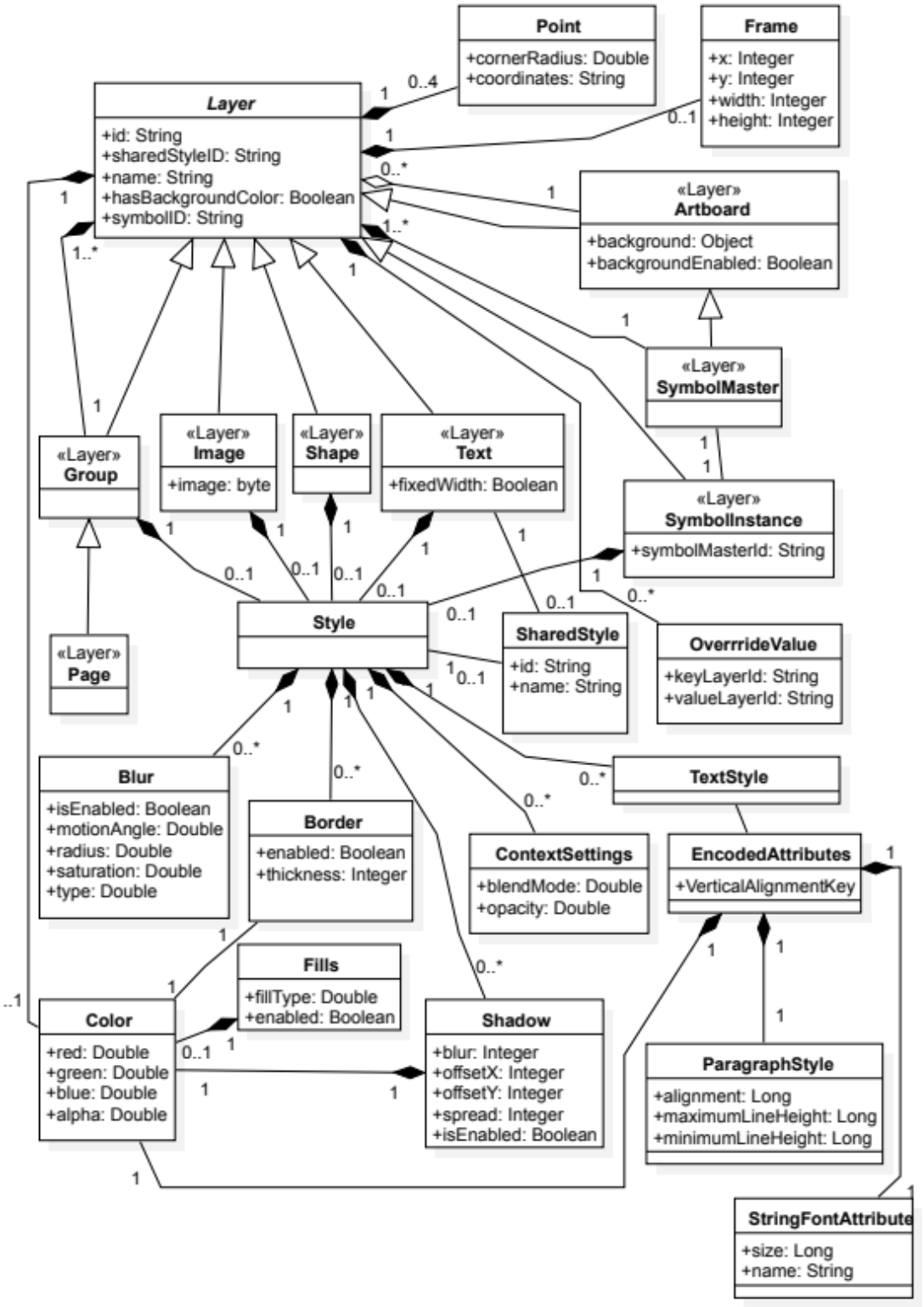


Figure 2.11: "Sketch Page Domain Meta-model"[16]

- **SymbolID:** If the Class attribute is a Symbol Instance, then it will have a SymbolID which corresponds to its Symbol Master.
- **Frame:** Defines the positions and dimensions of the layer relative to the parent layer.

2.3.2.2 OutSystems UI Template Sketch File Model

OutSystems' UX designers use a template whenever doing low-Fidelity mockups. This facilitates the mapping process later on, as it would not be possible to do so without having some guarantees of the design mockups' structure. This file was developed by OutSystems' UI team[17] and continuously gets updated in a way to match the evolution of the OutSystems UI framework. During the research, some issues around the usage of this file arose such as:

- Due to its continually evolving state, some designers are using different versions of this file.
- Some designers do not use the template at all.

The current template's structure is divided into the following Pages:

- **Style:** Represents the common styling properties such as typography, color, and shadow. Styling related concerns are completely out of scope of this work, so while this Page is highly useful to designers and developers, it is not for us.
- **UI Patterns and Widgets:** This page contains every UI pattern and widget in the OutSystems UI kit. These are all Symbol Instances and show all the different states of the components. This page is the one that shows different structuring throughout different versions of the template.
- **Layouts:** The page shows the different layouts of the pages for the different possible devices, such as a computer, tablet, mobile, and various screen sizes.
- **Symbols:** It contains every Symbol Master that's going to be used in the design. As mentioned before, one change in these symbols will propagate through their instances.

2.3.2.3 OutSystems Model

The OutSystems platform employs a Visual Programming Language that abstracts an underneath strongly typed language [91] that composes its four significant layers: **processes**, **interfaces**, **logic**, and **data** [78]. These layers can be seen in their IDE Service Studio (Figure 2.12).

In this work, we will focus on the interface layer since we are focusing on composing the UI's structure and, as the name suggests, this layer is related to the UI of an OutSystems project.

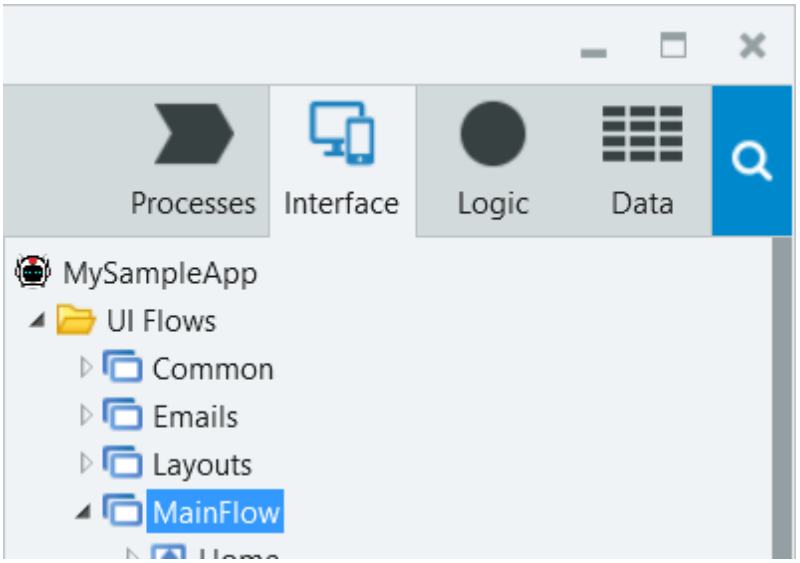


Figure 2.12: OutSystems' main 4 layers seen in Service Studio.

OutSystems' applications are composed of various modules to help reusability and maintainability. Each module is separated in its OML file. These files are proprietary to OutSystems and contain the information of modules. To see this information, you need a specialized tool created by OutSystems that extracts the information in these OML files into XML format. There are four kinds of modules: **application**, **service** [78], **library** [73], and **extension** [71]. The two kinds of modules I are interested in are the library and application modules.

The **library** module is pretty much self-explanatory as its goal is to contain the foundations of an application, such as its UI elements and Style Guides.

The **application** module contains the application itself. The application's screens are called **Web Screens** and are composed of various UI elements representing a hierarchical structure. As we can see, this already presents some similarities to the Sketch model. The mentioned UI elements can be separated into multiple categories, such as "widgets" whose primary goal is for the user to interact with the system by inputting or submitting. An example of these UI components are buttons and dropdown menus. To place these elements in an OutSystems application, the user drags and drops them into the pages.

These are a few similarities and mappings we can already do from one model to the other:

	Sketch	**OutSystems**
Internal Structure	Hierarchical Structure of Layers	Hierarchical Structure of Widgets
Application Screens	Artboard	Web Screen
Atomic Elements	Symbols	LSG Elements (UI Patterns)
Element Grouping	Groups	Containers or Web Blocks

Table 2.1: Mappings between Sketch and OutSystems

3 Related Work

The research on the automation of application creation from design inputs is relatively novel compared to the other methods of automatic generation from model based code generation [18, 30, 40, 57]. However, these past few years have shown some recent advances in this field. This chapter will present some relevant work in design to code translation, from state-of-the-art commercially available tools to more academic research.

3.1 Commercial Applications

3.1.1 Currently in the Market

There are several continuing efforts to minimize the gap between design and front-end development by converting the design into code.

Some existing applications, such as Anima [4], Supernova [95], and Yotako [99], allow users to provide their designs from multiple design tools such as Sketch [13] or Figma [27]. Most of these tools only export to a selection of traditional frameworks such as React or HTML/CSS. Other applications like React Studio [53], PaintCode [87], and PageDraw [81] are even more strict, as they require the design to be made in their built-in editor. This restricts designers from using their dedicated design application of choice and clashes with today's industry practices in the design field as verified by Bexiga [16]. None of them supports the generation of low-code artifacts.

While researching these tools, many were restricted to creating the same set of web technologies, with React being shared in almost all of them. None allowed for the generation of low-code artifacts, solely focusing on the current more traditional front-end technologies. Low-code platforms are a growing force in the market[92, 98] and just like these traditional technologies, they would benefit from the existence of a tool like these.

3.1.2 Emerging Applications

New commercial applications dedicated to transforming design artifacts into code are still emerging. An example of this is Modulz[60]. Modulz, by a startup of the same name,

differs from other design tools such as Sketch[13] or Figma[27]. Both Sketch and Figma are vector drawing tools, while Modulz is a dedicated UI and development tool, which means that when designers are creating their designs, the application is automatically generating code in the background. This auto-generation, as of right now, is only to React. Although they have plans to support more web technologies in the future, they suffer the same as the current market applications do by not supporting the low-code emerging platforms.

3.1.3 Plugins

Sketch and Figma are two of the most commonly used vector design tools globally [82, 83]. None of these applications allow natively the possibility of exporting the designs into code directly from the application. However, they allow expansion via plugins. Some of these plugins can be community-driven or even company-driven. One such example is their Code Generator plugin made by PaintCode [88]. This plugin allows the transformation from design to code to Objective-C, C#, and Swift.

Similarly, in Figma, a code generator technique was implemented [41] to convert the designs into React directly from the application. However, neither of these functionalities are employed by the professional teams in OutSystems since these code generation functionalities do not export to their low-code technology [17].

3.2 Methods to Identify UI Elements

One of the first and more well-known studies is Prefab. Done in 2010 by Dixon and Fogarty [24], Prefab uses computer vision to capture a window in a screen, interprets it, and recreates it in a new dummy window presented to the user. This dummy window is a copy of the original with added HCI accessibility techniques like the Bubble Cursor [33]. The user interacts with this dummy window, and the actions of the user are reflected in the original. Prefab relies on a predefined UI components model and assumes the pixels that make a widget are similar or even identical across various applications. This is not true in the current day and age as most applications have their styles and themes, and one cannot make assumptions on how the interface will end up looking. One of the limitations of this work is the relationship between components. Elements like a group of radio buttons or elements hidden in tabs or menus do not behave properly.

While Prefab supported the identification of User Interface elements, more recent approaches also support the direct transformation into code.

Nguyen et al. introduced in 2015 a novel proposal in the field of automatic application generation, REMAUI [63]. Unlike Prefab, REMAUI uses an interface's design as the input to generate an application's code, specifically from screenshots of the interface and does not require the complete model to already be defined. To our knowledge, they were the first to propose code generation through the UI design of applications' interfaces.

REMAUI uses a mixture of Optical Character Recognition with domain heuristics and computer vision techniques to achieve higher accuracy results while managing to preserve the structure of the application's pages.

Pix2code [15], two years after REMAUI, is another approach to the generation of applications from images. Beltramelli's model uses deep learning to generate an application's GUI in computer code based on a single input image. Unlike REMAUI, pix2code achieves its goal with a combination of Convoluted Neural Networks techniques and Long Short-Term Memory [32]. Beltramelli's tool outputs code with a 77% accuracy ratio with the original image for three platforms (iOS, Android, and web-based technologies). One of the significant drawbacks is related to using deep neural networks since it needs to do a lot of training to be better tuned. Nevertheless, his method managed to output code with 77% accuracy and managed to preserve the application's hierarchical structure.

Sketch2Code [39] generates code based on hand-drawn sketches. Their approach relies on methodologies that use sketch-based prototyping in the early phases of software development, which does not happen in our context. Their tool's results were highly dependent on good lighting when taking a photograph of the hand-drawn sketch, as bad lighting would bring in inaccurate results. They also managed to preserve the pages' structure, but unlike the previous, they could not identify the styling.

In 2020, Bexiga developed a method to automate the reusable UI components' styling process by transforming these components from an artifact created in a design tool into a low-code technology [16, 17]. Her goal was to mitigate another inefficiency in the collaboration between designers and developers, which makes it the most related to this work. Her work was also applied to the OutSystems ecosystem in the Customer Success department. The method she developed is focused on the styling properties of the reusable UI components. The developers' work was done manually by inspecting the designs and manually styling the components, and this tool allowed the automation of this process to shorten the time required. Her tool achieved results ranging from 20% to 70% time-saved for front-end developers, which could correspond to 2-3 days of work. Comparing her work to this one, I am also mitigating inefficiencies in the collaboration between these teams, and I am too applying it to the OutSystems ecosystem, but we are ultimately tackling distinct problems. Mine focuses on the composition of the reusable components to create application pages which will be applied to the second phase of the front-end development process, while hers applies to the first (these two phases were discussed in 2.1.6.2). Both of the works also differ in terms of their final goals. Her work is focused on shortening the time front-end developers spend on instantiating the reusable components while mine is to enable the development of more Sample Pages by the development team in the time they have now without extra investment.

These current approaches and others [34, 56] utilize images and hand-drawn sketches as their input making them somewhat disconnected from current established practices designers follow today. While people still use pen and paper to hand draw ideas during brainstorming sessions [83] or for initial testing using paper prototyping [64] it does

not go much further than this. These strategies are often employed to save money and resources during the early stages of the development process. However, these strategies are time-consuming and leave out essential details of the system such as looks, feedback, overall feel, and other HCI essential elements such as Fitts Law which states that the time to move to a target (in our case with a mouse or a finger in touch-based screens) is a function between the distance to the target and its width [20]. To get the system's feeling, look, and interactions, designers turn to these specialized design tools to construct a higher fidelity design that incorporates these elements. The table (3.1) summarizes this section.

3.3 Representation of User Interfaces

There are various languages created to specifically represent user interfaces *independently of physical characteristics*. These interfaces are called User Interface Description Language (UIDL) and are usually XML-based languages. In this section, I will focus on three examples of these UIDL, although more exist[49, 84, 89], since my solution will require an Intermediate Representation, that will technically be a UIDL.

User Interface Markup Language (UIML)[1] is a platform-independent language for describing UI. It is XML-based and is divided into five sections: description (where the components will be defined), structure(the structure of the UI), data (where is defined the content of the components), style (place to define the styling of the components), and events (section dedicated to defining the system events such as button presses). UIML has all of this to allow its' platform-independence nature, but some variations of UIML such as UIML/GUI[1] allow for a more compact architecture at the cost of a dependence (in this case, a graphical user interface). Due to its high level of abstraction, other UIDLs (such as the ones I will speak here) are often referred to UIMLs instead of UIDLs.

Extensible Application Markup Language (XAML)[58] is an XML-based declarative language created by Microsoft for their .NET applications. Since this language is the basis of Microsoft's .NET applications, it contains the structure of the User Interface and embeds programming logic and styling. Every XAML tag corresponds to a .NET component whose properties can be controlled through the tags' properties.

XML User Interface Language (XUL)[26] was created by the Mozilla Foundation. It is structured similarly to Web Pages. Like XAML, XUL can be extended with existing standards and technologies such as CSS and JavaScript.

These presented UIDLs (and others) were all created with the goal of representing UI in XML based representations and are specialized in frameworks. I will be needing a similar representation to these that fits more in the lines of the artifacts created by Design Tools and low-code front-end technologies. I could try to pick one of these and try to adapt them into our needs but ultimately I decided to create my own. My representation will differ from most UIDLs as it will be JSON-based instead of XML since I will be working with JSON throughout the process and have everything already setup.

Title	Input	Output	Styling	Structure	Notes
Prefab[24]	Screenshots	Creates a replica of the input window with enhancements.	✓	✓	Requires full model to be defined and counts on applications looking similar. Does not handle relationships between widgets.
REMAUI[63]	Screenshots	Mobile Technologies	✓	✓	Does not follow current industry practices.
pix2code[15]	Screenshots	Web & Mobile Technologies	✓	✓	Does not follow current industry practices.
Sketch2Code[39]	Hand-Drawn Sketches	Platform-Independent UI Representation	X	✓	Does not follow current industry practices.
Extraction and Classification of User Interface Components from an Image[34]	Screenshots	JSON file containing the structure of the UI	✓	✓	Does not follow current industry practices. Styling is very basic, limited to primary colors.
Closing the gap between designers and developers in a low code ecosystem[17]	Design Tools	Web Technologies	✓	X	The work focused on identifying the styling properties in the design artifacts provided to automate the styling process done by front-end professionals.
My Solution	Design Tools	Web Technologies	X	✓	Styling is dependent on the components provided. It does not interpret the colors from the given input.

Table 3.1: Related Work Summary: Academic Methods

4 Implementation

This chapter focuses on the approach I developed. It will showcase a high-level view of the process, the decisions needed to be made along the way (with respective discussion and reasonings) and conclude with a more in-depth view of the solution alongside some of the more noteworthy steps, such as the creation of my Intermediate Representation.

4.1 High-Level Solutions

My work focuses on the transformation from the source model Sketch into the OutSystems model. The solution takes as input a design file in Sketch that contains the pages of the application and the components instantiated in OutSystems. It generates an OutSystems application with fully structured screens and a report file containing user info such as errors and patterns that were not converted. A representation of the basic concept can be seen in Fig.4.1.

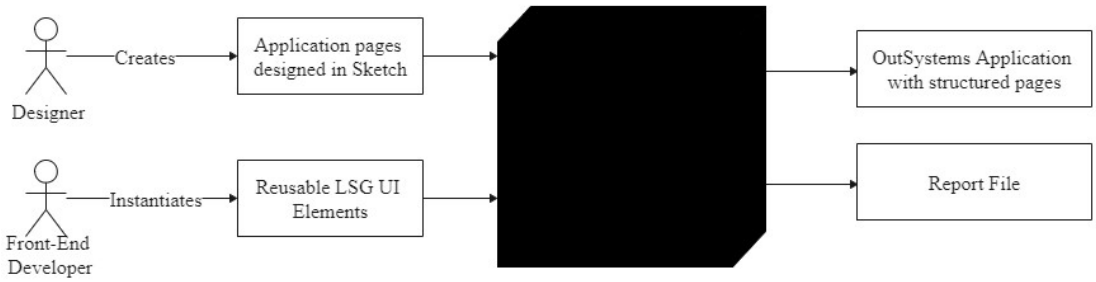


Figure 4.1: High level overview of my proposal

When composing a jigsaw puzzle, the player relies on the image given, usually in the front of the box, to know where to place each piece to recreate the image. This solutions' overall workflow can be seen in a similar light to mounting a puzzle. Each application page in the Sketch file can be seen as a puzzle image, while the reusable LSG elements are the puzzle pieces. The tool receives the application's pages in Sketch to understand the overall structure, the components that compose its structure, and its relationships.

Knowing each page's structure, the tool uses the provided reusable LSG UI elements to recreate the page in the OutSystems language by composing the pages using these reusable elements.

Considering the insights gathered from the research and interviews carried out with the design and developer teams (section 2.1.6 shows the information gathered from these interviews), there were two possible solutions. These solutions are fundamentally similar but differ in one important factor: which type of design artifacts it uses, high-fidelity or low-fidelity. Each has its pros and cons as its structure differs.

4.2 Low-Fidelity vs High-Fidelity

4.2.1 High-Fidelity

Looking first at the high-fidelity approach, I propose using the high-fidelity mockups created at the end of the UI phase to compose with the UI components already in OutSystems. This solution has the advantages of creating fully structured pages, the Front-End team would suffer no impacts to their workflow, and that the tool would be generating pages closer to the final product since after these were done, the design process was considered finished (besides a few exceptions). However, on the Design side of the process, there would be required adjustments to their workflow, so they bootstrap the LSG while working on the pages to mitigate the decoupling issue mentioned in section 2.1.6.1. Table 4.1 shows a summary of this approach.

Pros	Cons
Pages are completely structured.	Adjustments to the current design workflow would be required.
Front-End developers would suffer no impact to their current workflow.	
Pages resemble more closely the final product.	

Table 4.1: Pros and cons of the High-Fidelity approach

4.2.2 Low-Fidelity

Instead of using high-fidelity mockups, this approach proposed using the mockups created during the UX phase where the page's structure would already be complete while missing all the styling such as colors, margins, paddings, etc. This option continues producing completely structured Sample Pages, now both the Front-End and Design workflows would suffer no impact, and we gain a new advantage of having a higher potential of more Sample Pages since the low-fidelity mockups have a higher number of pages than the high-fidelity. Here, we are also more safe from the decoupling issue since

the established practice during the construction of these designs dictates that the designers should use the components in the template. The disadvantage of this approach is that it is viable to ambiguity issues. Since this approach runs on a higher abstraction level, a component in the low-fidelity mockup of a page could map to more than one entity in the OutSystems language components due to variations, i.e., one button in low-fidelity could have five variations. Table 4.2 shows a summary of this approach.

Pros	Cons
Pages are completely structured.	Ambiguity risk - How do we differentiate variations?
Front-End developers & designers would suffer no impact to their current workflow.	
Low Fidelity mockups have more pages.	
They are always based on a template.	

Table 4.2: Pros and cons of the Low-Fidelity approach

4.2.3 Ambiguity Challenge

To evaluate the ambiguity risk presented in the previous section, I checked 20 past OutSystems projects to see if the issue presented itself and, if so, how often. These projects were divided equally into one of four complexity levels. This research found the risk to be minimal as none of the projects possessed any ambiguity in their elements when shifting from low-fidelity to high-fidelity, which coincides with information passed in the interviews. What usually happens to handle variations is: either the template has already a component per variation (for example being the button, there is a primary button component, cancel button, and a loading button) or Sketch allows the designers to modify an instance of the component while preserving the connection to the template (for example, the primary button in the template is blue and the designer can turn the instance red while preserving the connection to the original primary button).

4.2.4 Conclusion

After verifying that the ambiguity risk presented to be minimal, I decided to focus on using the Lo-Fi mockups for my work while completely ruling out Hi-Fi. This decision presented to be not ideal, as most projects **do not** preserve the UX design, or they simply skipped that phase. I could not ignore the fact that the volume of projects with Hi-Fi mockups was significantly larger. With this in mind, the final work ended supporting both cases, with the caveat being that when using high-fidelity we have less guarantees that the component identification will be as successful as when using low-fidelity.

4.3 Implementation Design Decisions

At this point, there were set a few other decisions that would shape the entire implementation process, the first being a set of requirements I believed to be essential for this work. They were as follows:

- The tool should be as unintrusive as possible, to facilitate its adoption among UX/UI designers, so as not to disrupt their workflow, while providing a significant head-start for front-end developers, freeing their time.
  - If any adjustments are needed, these should be minimal and not intrusive.
- The tool should be simple to use.

The other decision was how aggressive we should be with the automation process. Here, I came up with two opposing approaches:

In the first approach, I would focus on **Precision**, by only instantiating the components, and parts of them, when and where we are close to 100 % sure we are identifying them correctly. The consequence of this approach is that we end up with fewer components, and the resulting pages would be more incomplete with the upside of having a significantly improved accuracy.

The second approach would focus on **Recall** by automating as much as possible, even for ambiguous scenarios when we are no 100% sure of the appropriate process. This leads to a higher risk of incorrectly instantiated components, forcing the development team to reorganize or, in the extreme case, redo from scratch.

I decided and opted to focus on **Precision** with approach 1 since it causes fewer overheads on the development team. I am focusing on automating and accelerating part of their day-to-day workflow, and so purposely giving them extra work would be against my philosophy, and so I decide it is better to deliver an incomplete result that they can quickly complete with their expertise. We also diminish the possibility of potential frustration from part of the development team due to errors in the transformations.

4.4 Intermediate Representation Model

I created my own model to bridge our source and target models to achieve a higher abstraction level. My model is mainly based on the Sketch model, with additional information to support a correct mapping to OutSystems.

The base element is an IRNode, an abstract structure created by us to serve as the nodes of my Intermediate Representation, which follows a tree-like structure. There will be an IRNode per supported Sketch layer/OutSystems component, containing all the information that is common in every component, such as:

- **Type:** Represents the type of the node. It can be a group, symbol, or root, determined by the *Class* of the layer.
- **InternalName:** A string that represents the name of the component in the Live Style Guide.
- **Widget:** A simplified version of InternalName. It allows to easily distinguish components, i.e., an InternalName like 01.adaptable/component/Button Default/Primary goes to Primary Button.
- **DesignerName:** The layer name, as chosen by the designer.
- **RelativePosition:** Contains the coordinates of the component in relation to the parent component, its width and height.
- **Children:** A list containing the descendant layers. Useful for nested components.
- **Visited:** A Boolean value that exists to help calculate overlaps - if a component overlaps with another in the x-axis - without duplicated counting.

I created support for 20 components of the OutSystems LSG. These components were identified by the Front-End experts at OutSystems as the most relevant components. The following Figures 4.2, 4.3, 4.4 and 4.5 showcase the class diagrams of the implemented components.

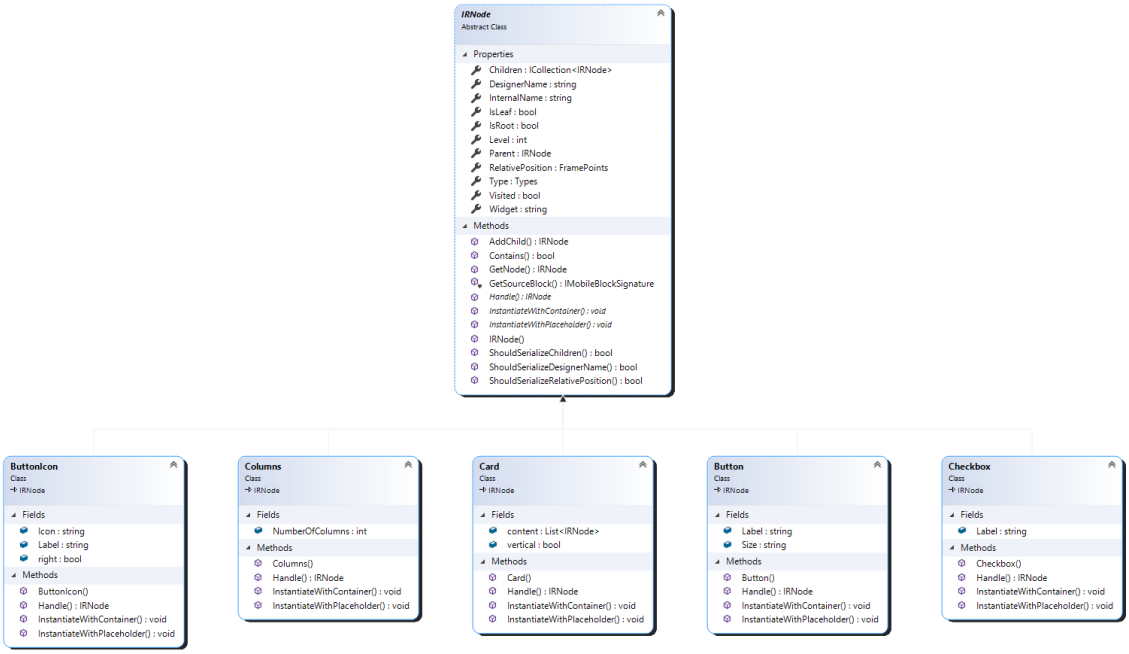


Figure 4.2: Class diagrams for ButtonIcon, Columns, Card, Button, Checkbox components.

Note that each component has Fields that are particular for that component and all of them possess the 3 main methods:

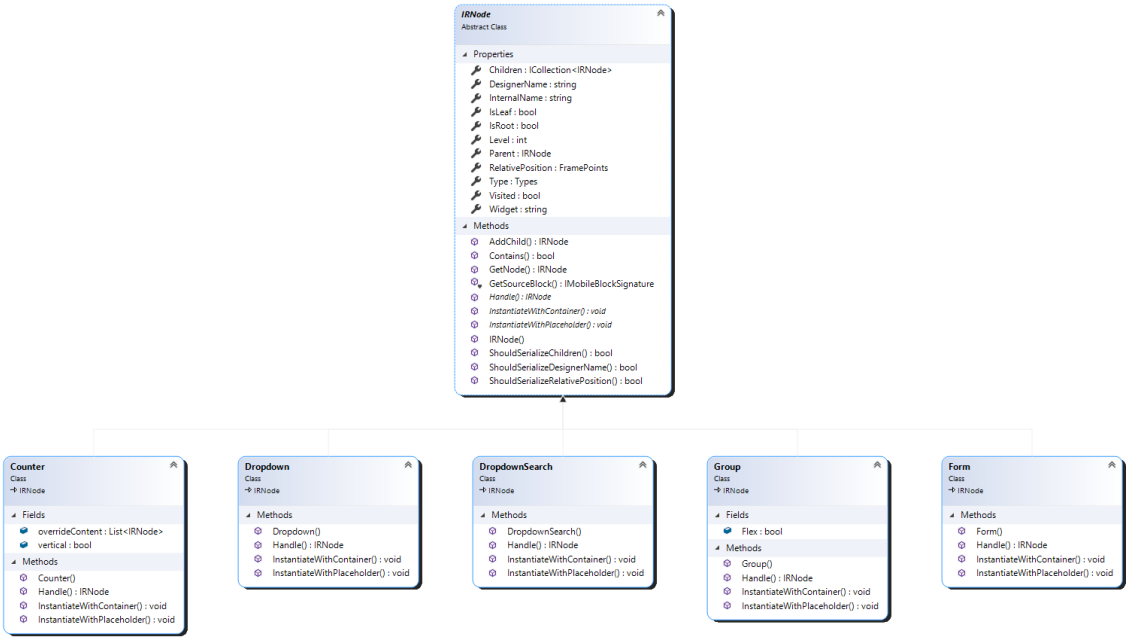


Figure 4.3: Class diagrams for Counter, Dropdown, DropdownSearch, Group and Form components.

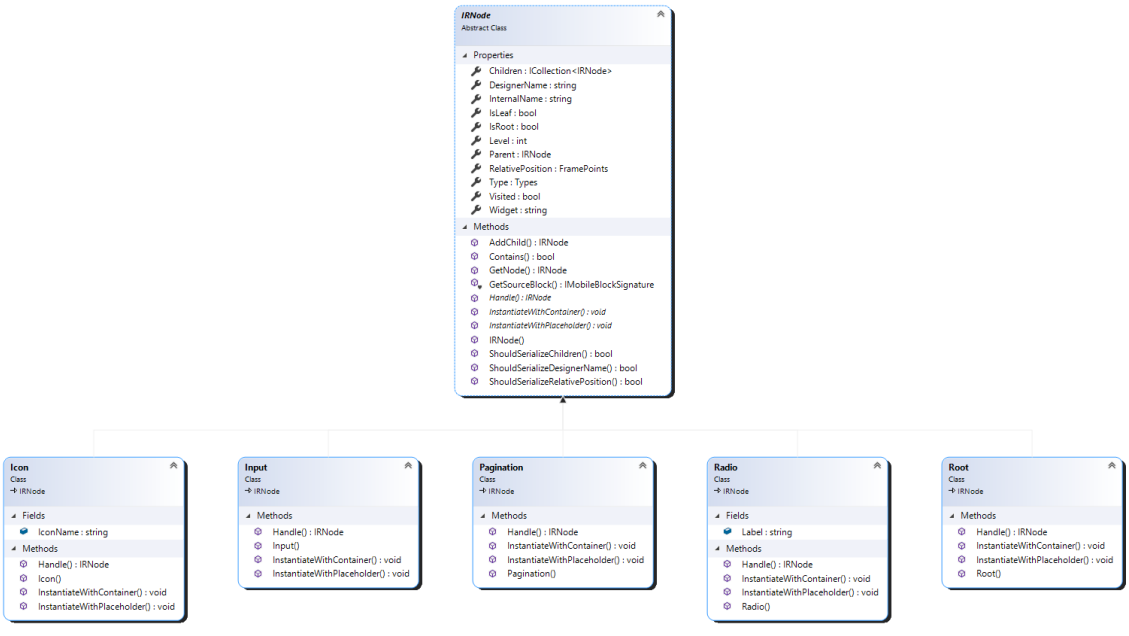


Figure 4.4: Class diagram for Icon, Input, Pagination, Radio and Root components.

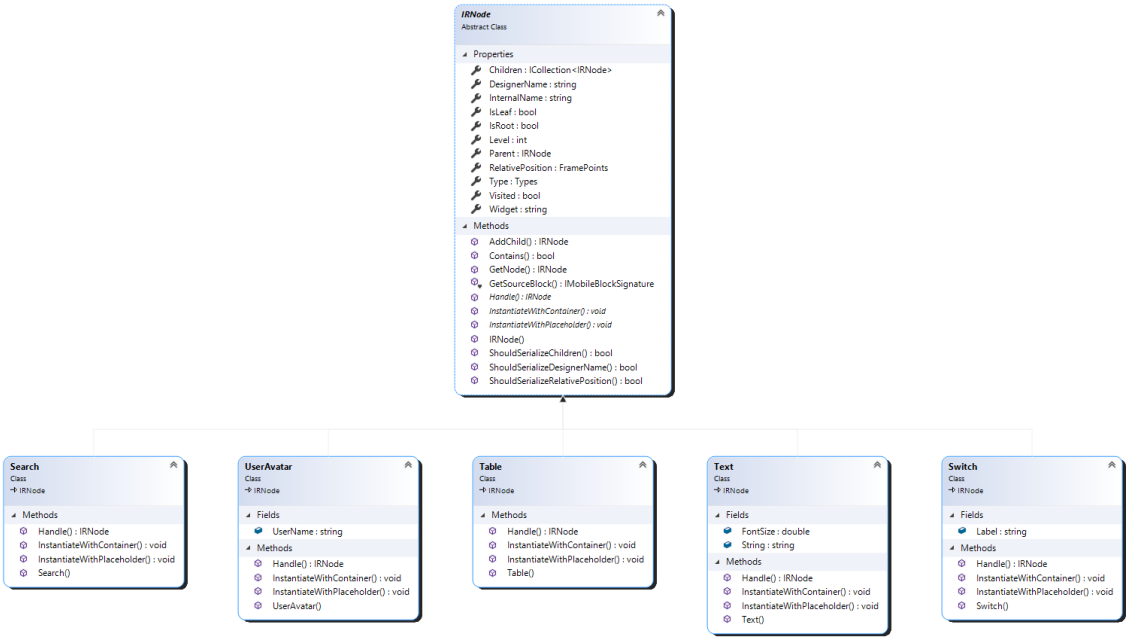


Figure 4.5: Class diagram for Search, UserAvatar, Table, Text and Switch components.

- Handle: Used in the first transformation from Sketch to my Intermediate Representation. Used to populate the Node with the required info from the Layer.
- The following 2 are for the second transformation, from Intermediate Representation to OutSystems:
  - InstantiateWithContainer: Instantiate the component in given container.
  - InstantiateWithPlaceholder: Instantiate in the given placeholder.

With the Intermediate Representation established, a more zoomed in approach can be seen in Figure 4.6. This figure showcases the two big phases that my work follows.

The first major phase is a transformation from Sketch (source model) to my Intermediate Representation (target model). This phase will receive a Design artifact in Sketch created by a designer in the design team, and create 1 file per screen mockup.

The second phase will be the composition phase, where the tool will receive the screens in my Intermediate Representation (**source model**) and a library of reusable UI components already instantiated and stylized in OutSystems by a developer, and compose both together to deliver an OutSystems (target model) application with structured pages.

4.5 Sketch To Intermediate Representation

To handle the Sketch model, I will be proceeding with the unzipping method, where we unzip the .sketch file and receive multiple JSON files to work with. Another possibility would be to use Sketch's Command Line Interface(CLI) [8] or Sketch's API [7]. These approaches rely on having an installation of Sketch to access as they come bundled with

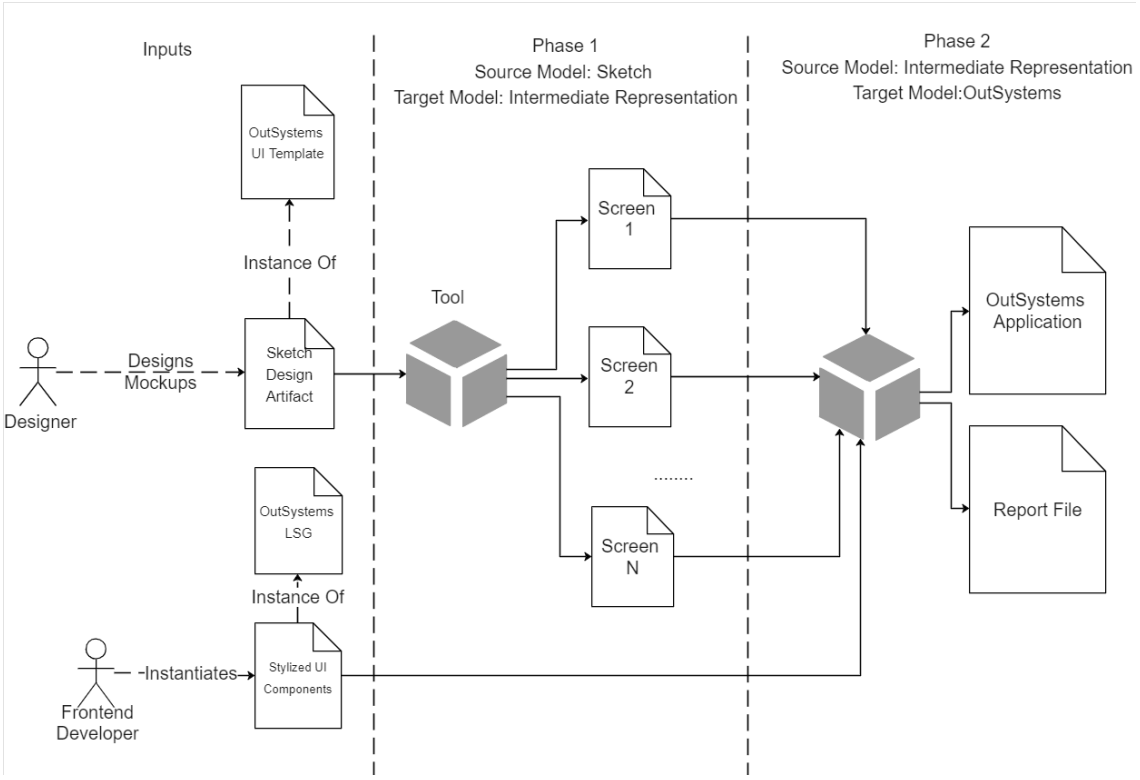


Figure 4.6: Zooming in the approach.

the CLI, which in turn requires the user to have the software and the hardware as Sketch is only available on Apple computers at the time of writing. Since OS X only represents 16% of the market share, as seen in Figure 4.7, these approaches present themselves as more restrictive than I would like.

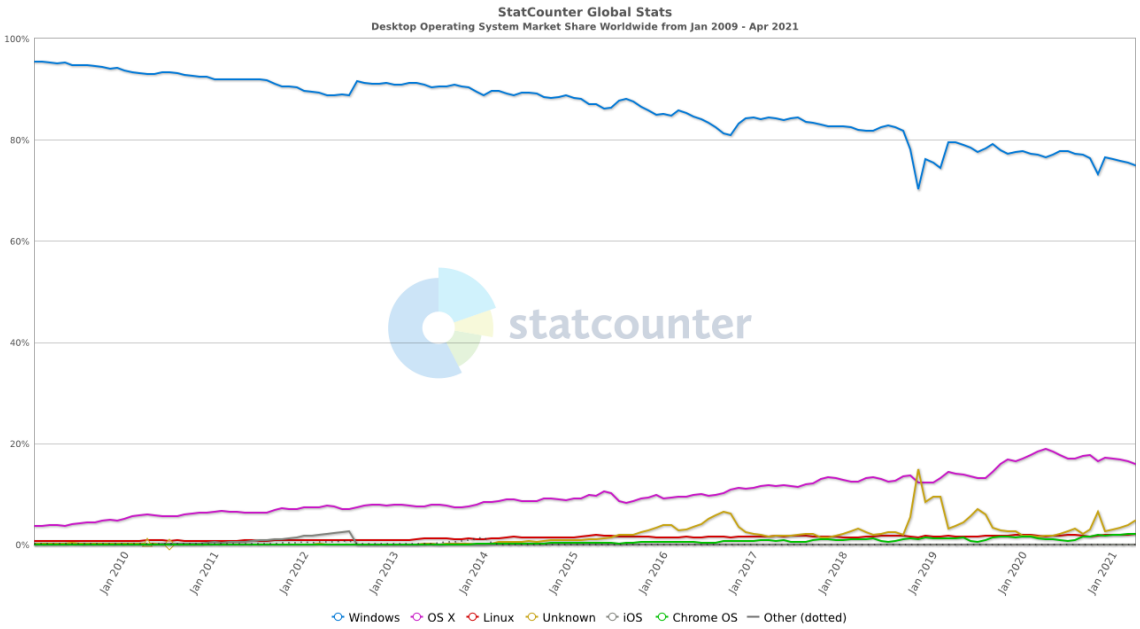


Figure 4.7: Operating Systems market share [94].

On the other hand, the unzipping method works in almost, if not all operating systems and so, I decided to adapt this approach.

4.5.1 Initial Process

To access and manipulate a Sketch file, we first need to unzip the file as documented in the Sketch documentation [9]. By unzip the file, it results in three JSON files and three folders. They are as follows:

- **meta.json:** Contains the metadata of the Sketch file such as a list of the *Pages*, Sketch version and fonts used. For my work, I do not use this metadata file.
- **document.json:** Contains common data that is shared between all *Pages*, such as shared styles. Not relevant to us as it partakes more in styling issues.
- **user.json:** Contains the user data for the Sketch file, like zoom level, dimensions, UI metadata and if it has been uploaded to the Sketch Cloud. This file mostly contains settings at the user level which is not relevant to us.
- **pages folder:** This folder contains a JSON per *Page* in the Sketch file. This folder is the most relevant for us as the contents of the files within this folder contain the internal structure of the pages in the project.
- **images folder:** The images folder is self explanatory, in it we can find all the bitmaps that were used in the project at their original scales. Since I do not handle images, we can ignore this folder.
- **preview folder:** This folder only contains an image of the last page edited.

After the unzipping process, now I needed to find the JSON files that correspond to the *Pages* we want to work with. Namely, we are interested in the *Pages* that contain the screen's designs and the Symbol *Page* that contains all the Symbol Masters. Since we cannot look at the file's names to identify the Pages, as these are generally a string of random characters, we need to look at the inner structure of the JSON files. Specifically, we need to inspect the first **Name** property of the top-level Layer as it will match the name given by the designer. The Page that contains the Symbol Masters is trivial to identify, as this Page is always called (at the current version of the template) **Symbols**. The Page that relates to the mockups, however, is not so simple. A designer creates this Page, and so the page is susceptible to an arbitrary name. With this in mind, the Pages require to have consistency in their names. Here, we look for pages containing the string "UX Design" or "UI Design" in their names. If only one exists, I will use that Page. If both exist, I have a priority method to prioritize the UX Design over the UI Design for reliability concerns, as it is an established practice to use the template's components to model this kind of design, allowing us to identify the components with high precision (plus other reasons mentioned in section 4.2).

4.5.2 JSON Treatment

Now that we have correctly identified the JSON files to use, we needed to find a way to manipulate them. These JSON files are composed of Layer objects and are full with a large number of these objects. An example can be seen in Fig 4.8 which is the JSON corresponding to the Symbol Master Page. A quick look at the image or by inspecting the scroll bar on the right, it is possible to see how much information these files contain. Although here we present a Page corresponding to a Symbols' Page, the ones that correspond to the UX or UI Design have similar lengths and complexity, if not more.


{"_class":"page","do_objectID":"6C7B0E88-D586-459B-B156-2EF7FFE2DDE5","booleanOperation":-1,"isFixedToViewport":false,"isFlippedHorizontal":false,"isFlippedVertical":false,"isLocked":false,
"isVisible":true,"layerListExpandedType":0,"name":"Symbols","nameIsFixed":true,"resizingConstraint":63,"resizingType":0,"rotation":0,"shouldBreakMaskChain":false,"userInfo":{"com.animaapp.
stc-sketch-plugin":{"com.animaapp.uses.anima":1,"kModelPropertiesKey":{}},"com.betterthantomorrow.sketch.distributor":{"distributorDimension":1,"distributorSpacing":"20","distributorCenterSpacing":0},
"com.sonburn.sketchplugins.symbol-organizer":{"globalSettings":1}},"exportOptions":{"_class":"exportOptions","includedLayerIds":[],"layerOptions":0,"shouldTrim":false,"exportFormats":[],"frame":
{"_class":"rect","constrainProportions":false,"height":0,"width":0,"x":0,"y":0},"clippingMaskMode":0,"hasClippingMask":false,"style":{"_class":"style",
"do_objectID":"AFFEE6A0-7094-4E9F-B3B0-76095E6EACC0","endMarkerType":0,"miterLimit":10,"startMarkerType":0,"windingRule":1,"blur":{"_class":"blur","isEnabled":false,"center":"{0.5, 0.5}",
"motionAngle":0,"radius":10,"saturation":1,"type":0},"borderOptions":{"_class":"borderOptions","isEnabled":true,"dashPattern":[],"lineCapStyle":0,"lineJoinStyle":0,"borders":[],"colorControls":
{"_class":"colorControls","isEnabled":false,"brightness":0,"contrast":1,"hue":0,"saturation":1},"contextSettings":{"_class":"graphicsContextSettings","blendMode":0,"opacity":1},"fills":[],
"innerShadows":[],"shadows":[]},"hasClickThrough":true,"groupLayout":{"_class":"MSImmutableFreeformGroupLayout"},"layers":[{"_class":"symbolMaster","do_objectID":"BE42C16A-A932-4788-B789-B52D58FF61B1",
"booleanOperation":-1,"isFixedToViewport":false,"isFlippedHorizontal":false,"isFlippedVertical":false,"isLocked":false,"isVisible":true,"layerListExpandedType":1,"name":"03. Interaction ⋁ 02.
Animated Label ⋁ Box ⋁ Active","nameIsFixed":true,"resizingConstraint":63,"resizingType":0,"rotation":0,"shouldBreakMaskChain":true,"userInfo":{"com.animaapp.stc-sketch-plugin":
{"kModelPropertiesKey":{}}},"exportOptions":{"_class":"exportOptions","includedLayerIds":[],"layerOptions":0,"shouldTrim":false,"exportFormats":[],"frame":{"_class":"rect",
"constrainProportions":false,"height":50,"width":200,"x":25232,"y":206},"clippingMaskMode":0,"hasClippingMask":false,"style":{"_class":"style","do_objectID":"C3890E8D-D02D-4D98-937E-38701FF90709",
"endMarkerType":0,"miterLimit":10,"startMarkerType":0,"windingRule":1,"blur":{"_class":"blur","isEnabled":false,"center":"{0.5, 0.5}","motionAngle":0,"radius":10,"saturation":1,"type":0},
"borderOptions":{"_class":"borderOptions","isEnabled":true,"dashPattern":[],"lineCapStyle":0,"lineJoinStyle":0},"borders":[],"colorControls":{"_class":"colorControls","isEnabled":false,"brightness":0,
"contrast":1,"hue":0,"saturation":1},"contextSettings":{"_class":"graphicsContextSettings","blendMode":0,"opacity":1},"fills":[],"innerShadows":[],"shadows":[]},"hasClickThrough":true,"groupLayout":
{"_class":"MSImmutableFreeformGroupLayout"},"layers":[{"_class":"symbolInstance","do_objectID":"EE1F6AE6-7CE6-43EA-9C98-CE2580DD687A","booleanOperation":-1,"isFixedToViewport":false,
"isFlippedHorizontal":false,"isFlippedVertical":false,"isLocked":false,"isVisible":true,"layerListExpandedType":1,"name":"Input Background","nameIsFixed":true,"resizingConstraint":47,"resizingType":0,
"rotation":0,"shouldBreakMaskChain":false,"userInfo":{"com.animaapp.stc-sketch-plugin":{"kModelPropertiesKey":{}}},"exportOptions":{"_class":"exportOptions","includedLayerIds":[],"layerOptions":0,
"shouldTrim":false,"exportFormats":[],"frame":{"_class":"rect","constrainProportions":false,"height":50,"width":200,"x":0,"y":0},"clippingMaskMode":0,"hasClippingMask":false,"style":{"_class":"style",
"do_objectID":"485E8AE6-82B9-4F40-842A-FA0201BE036A","endMarkerType":0,"miterLimit":10,"startMarkerType":0,"windingRule":1,"blur":{"_class":"blur","isEnabled":false,"center":"{0.5, 0.5}",
"motionAngle":0,"radius":10,"saturation":1,"type":0},"borderOptions":{"_class":"borderOptions","isEnabled":true,"dashPattern":[],"lineCapStyle":0,"lineJoinStyle":0},"borders":[],"colorControls":
{"_class":"colorControls","isEnabled":false,"brightness":0,"contrast":1,"hue":0,"saturation":1},"contextSettings":{"_class":"graphicsContextSettings","blendMode":0,"opacity":1},"fills":[],
"innerShadows":[],"shadows":[]},"horizontalSpacing":0,"scale":1,"symbolID":"5BD096FA-868A-4A7A-B8C6-CAEAABC8A2B1","verticalSpacing":0,"overrideValues":[{"_class":"overrideValue",
"overrideName":"CB81353F-0298-42B9-8A13-E80C6A2F591C_layerStyle","value":"F3283828-EC31-41BC-85FD-1A41E94F2AF3"}],{"_class":"text","do_objectID":"CB839479-2CC8-4A21-89E1-E7ABC614D8C2",
"booleanOperation":-1,"isFixedToViewport":false,"isFlippedHorizontal":false,"isFlippedVertical":false,"isLocked":false,"isVisible":true,"layerListExpandedType":0,"name":"Placeholder",
"nameIsFixed":true,"resizingConstraint":11,"resizingType":0,"rotation":0,"shouldBreakMaskChain":false,"userInfo":{"com.animaapp.stc-sketch-plugin":{"kModelPropertiesKey":{"isAutoWidth":1}}},
"exportOptions":{"_class":"exportOptions","includedLayerIds":[],"layerOptions":0,"shouldTrim":false,"exportFormats":[]},"frame":{"_class":"rect","constrainProportions":false,"height":21,"width":79,
"x":16,"y":25},"clippingMaskMode":0,"hasClippingMask":false,"sharedStyleID":"19CC9AD2-4B87-4D9A-A67A-B50065AB0C35","style":{"_class":"style","do_objectID":"311AAE58-0C2F-4063-8C7B-17A64F3C0CBE",
"endMarkerType":0,"miterLimit":10,"startMarkerType":0,"windingRule":1,"blur":{"_class":"blur","isEnabled":false,"center":"{0.5, 0.5}","motionAngle":0,"radius":10,"saturation":1,"type":0},
"borderOptions":{"_class":"borderOptions","isEnabled":true,"dashPattern":[],"lineCapStyle":0,"lineJoinStyle":0},"borders":[],"colorControls":{"_class":"colorControls","isEnabled":false,"brightness":0,
"contrast":1,"hue":0,"saturation":1},"contextSettings":{"_class":"graphicsContextSettings","blendMode":0,"opacity":1},"fills":[],"innerShadows":[],"shadows":[],"textStyle":{"_class":"textStyle",
"encodedAttributes":{"MSAttributedStringFontAttribute":{"_class":"fontDescriptor","attributes":{"name":"SFProText-Regular","size":14}},"MSAttributedStringColorAttribute":{"_class":"color","alpha":1,
"blue":0.47058823529411176,"green":0.44313725490196078,"red":0.41568627450980392},"textStyleVerticalAlignmentKey":0,"paragraphStyle":{"_class":"paragraphStyle","alignment":0,"maximumLineHeight":21,
"minimumLineHeight":21},"verticalAlignment":0}},"attributedString":{"_class":"attributedString","string":"Placeholder","attributes":[{"_class":"stringAttribute","location":0,"length":11,"attributes":
{"MSAttributedStringFontAttribute":{"_class":"fontDescriptor","attributes":{"name":"SFProText-Regular","size":14}},"MSAttributedStringColorAttribute":{"_class":"color","alpha":1,"blue":0.
47058823529411176,"green":0.44313725490196078,"red":0.41568627450980392},"textStyleVerticalAlignmentKey":0,"paragraphStyle":{"_class":"paragraphStyle","alignment":0,"maximumLineHeight":21,
"minimumLineHeight":21}}}],"automaticallyDrawsOnUnderlyingPath":false,"dontSynchroniseWithSymbol":false,"glyphBounds":"{{1, 4}, {77, 12}}","lineSpacingBehaviour":2,"textBehaviour":0},{"_class":"text",
"do_objectID":"83987D3C-6196-4FA0-9260-F752A17A1886","booleanOperation":-1,"isFixedToViewport":false,"isFlippedHorizontal":false,"isFlippedVertical":false,"isLocked":false,"isVisible":true,
"layerListExpandedType":0,"name":"Text","nameIsFixed":true,"resizingConstraint":11,"resizingType":0,"rotation":0,"shouldBreakMaskChain":false,"userInfo":{"com.animaapp.stc-sketch-plugin":
{"kModelPropertiesKey":{"isAutoWidth":1}}},"exportOptions":{"_class":"exportOptions","includedLayerIds":[],"layerOptions":0,"shouldTrim":false,"exportFormats":[]},"frame":{"_class":"rect",
"constrainProportions":false,"height":21,"width":28,"x":16,"y":25},"clippingMaskMode":0,"hasClippingMask":false,"sharedStyleID":"924E0BBD-E627-4D3A-8366-6BA101877867","style":{"_class":"style",
"do_objectID":"7EA27E501-3UB8-4UDD-B59C-3B2E91A12625","endMarkerType":0,"miterLimit":10,"startMarkerType":0,"windingRule":1,"blur":{"_class":"blur","isEnabled":false,"center":"{0.5, 0.5}",
"motionAngle":0,"radius":10,"saturation":1,"type":0},"borderOptions":{"_class":"borderOptions","isEnabled":true,"dashPattern":[],"lineCapStyle":0,"lineJoinStyle":0},"borders":[],"colorControls":
{"_class":"colorControls","isEnabled":false,"brightness":0,"contrast":1,"hue":0,"saturation":1},"contextSettings":{"_class":"graphicsContextSettings","blendMode":0,"opacity":1},"fills":[],
"innerShadows":[],"shadows":[],"textStyle":{"_class":"textStyle","encodedAttributes":{"MSAttributedStringFontAttribute":{"_class":"fontDescriptor","attributes":{"name":"SFProText-Regular","size":14}},
"MSAttributedStringColorAttribute":{"_class":"color","alpha":1,"blue":0.18823529411764471,"green":0.16862745098039222,"red":0.15294117647058822},"textStyleVerticalAlignmentKey":0,"paragraphStyle":
{"_class":"paragraphStyle","alignment":0,"maximumLineHeight":21,"minimumLineHeight":21}},"verticalAlignment":0}},"attributedString":{"_class":"attributedString","string":"Text","attributes":[
{"_class":"stringAttribute","location":0,"length":4,"attributes":{"MSAttributedStringFontAttribute":{"_class":"fontDescriptor","attributes":{"name":"SFProText-Regular","size":14}},

Figure 4.8: Example of Symbols Page

So to be able to manipulate these files easily I used a framework called JSON.NET [62]. This framework allows us to deserialize JSON objects into any given class, transforming these JSON files into trees with nodes of a class of our choosing.

4.5.2.1 Selecting Attributes

An option would be to serialize the JSON objects into a class that represented the standard Sketch Model. IDEs such as Visual Studio allow the easy creation of a class that models a JSON file by pasting said JSON into the IDE and clicking a button. A problem with this approach is that a Layer has an extensive list of attributes that are not necessary to us (list can be viewed in Fig. 2.10). So, I created a new class called "Artboard," where I selected the attributes that I wanted to preserve. Using JSON.NET by simply naming our properties the same way as the attributes in the JSON file will automatically correlate and serialize those fields with the correct values. Diagram 4.9 shows this simplification process.

The Pages, especially the ones related to the mockups, usually have more than just the Artboards. Usually, the designer adds more to the designs to better identify by giving

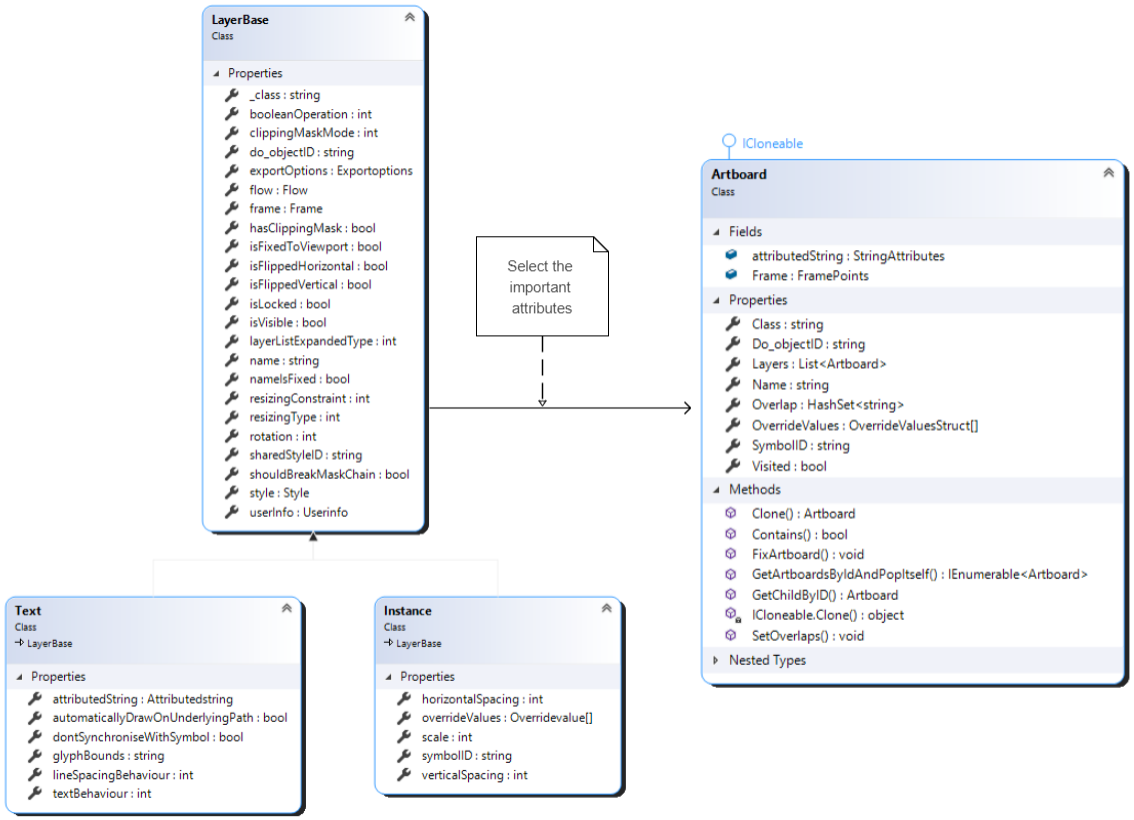


Figure 4.9: Transformation from Layer to my custom class Artboard

the artboards a name over them or with arrows to represent the flow of the screens. An example of both of these auxiliary artifacts can be seen in Figure 4.10. These elements from the Pages are not part of the Artboard and are simply auxiliary, so we ignore and skip them to solely keep the Artboards.

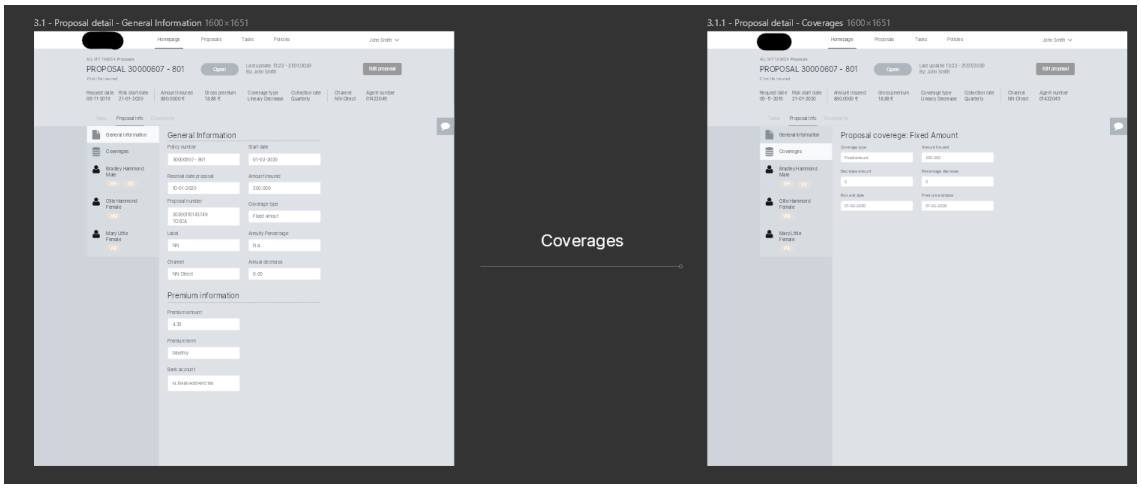


Figure 4.10: Example of a Page with 2 Artboards and auxiliary text and arrows

After this step is done, we have a tree per Artboard composed only by objects of my custom class, with all the properties we need.

From now on, **all Layers** of the process have this simplified model.

4.5.3 Component Identification

As we iterate through a tree corresponding to an Artboard, we need to identify the node we are looking at with the highest precision and certainty we can. To do this, we have three distinct possibilities.

The first possibility is exclusive to Symbol Instances. These Symbols have an attribute called "SymbolID" that links back to the Symbol Master. With this link between Instance and Master, we can quickly identify the component correctly by tracing back to the Master, part of the OutSystems UI template. The only problem with this approach is that it relies on the design team to use the Symbols as much as possible, which usually does not happen. In some cases, designers can see fit to create the components from scratch by hand since they are easier to create than to Instantiate (like the Card can be easily replicated by just creating a Group composed by a white rectangular Layer, then looking and searching for the Card Symbol) or that the instance would require some heavy modifications that Sketch generally does not allow them to do.

The second possibility is to identify a component by its composition, for example, identifying a blue underlined text Layer as a Link or a group with X number of input fields and buttons as a Form. I discarded this approach as I cannot guarantee that I am identifying these elements correctly. In the example given, that text Layer could be a Link, or it could be some particular styling choice that the designer decided. This possible ambiguity goes against my Precision first approach.

The third possibility is to look at the Layer's name. This approach can be precarious as Layer names are subjective to the designer. To adopt this approach, I would require some guarantees to have a more consistent result. To this end, I have adopted a method to identify Layers - all of them being in the Group class - by matching their names with the ones from the Symbol Masters. I restrict this verification to the Group class as **neither** the shape or text groups could be a UI Component. The Layers will be referred to as *Decoupled Components*.

The implemented program identifies the UI components based on the first and third approaches presented. These two approaches focus on tracing back to the components in the OutSystems UI Template, either by SymbolId in the first approach or by matching with their name in the case of the third one. This meant I needed to keep a reference of components in the template, namely the ones present in the Symbols Page (2.3.2.2). One possibility could be to store this information in a file (in JSON format, for example) that would map an ID to the Symbol Master. The problem is that this template is volatile, and there exist multiple versions of it already and will continue getting updated as time goes on, which would require this file to get updated as well or even have one version per template version. This would be highly impractical, and so I decided to build a structure that would accomplish this same feat, but that would be built on runtime.

4.5.3.1 Auxiliary Dictionary

This structure took the form of a dictionary that runs over the Symbols Page at the start of the program and adds entries, where the key is their ID, and the value is a structure containing the necessary attributes. Fig 4.11 showcases the Class created to be the values of my dictionary. We save the Name property from supporting the Decoupled Component solution. We also save the Layers property, which are the sub-Layers of the item. These are necessary when we need to instantiate components that do have any alteration.

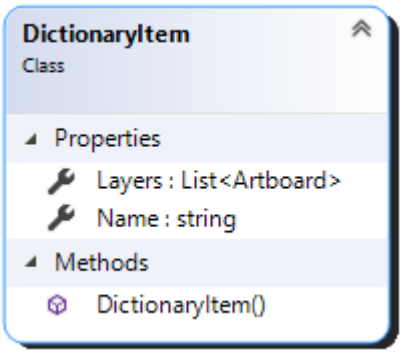


Figure 4.11: Class Diagram for the dictionary

With this dictionary, Symbol Instances can find the corresponding Master easily by looking up the ID which is the key, and Decoupled Components can find the correspondent by name, using LINQ over the dictionary's values.

4.5.4 IRNode Creation

Whenever an LSG UI component has been identified from a Layer, this Layer will be transformed into an IRNode. The Layer can have different types, but here we only handle 3 types: SymbolInstance, Group and Text. Algorithm 1 showcases an overview of the process.

4.5.5 Transformation Rules

From now until the end of this section, I will be showcasing the transformation rules for each of the components. These rules were hard-coded into the tool itself instead of utilizing a model transformation language. The rules that were created will be described by text followed by an equation that represents it. These equations follow the following format:

$$RuleName: \frac{LeftHandSide/Source}{RightHandSide/Target} \tag{4.1}$$

Both the sides of the equation can be extended to showcase a more detailed representation.

Algorithm 1 Overview of the Intermediate Representation creation

Data: An Artboard Layer with every sub-Layer already simplified
Result: A tree of IRNode

```
currentNode ← newRoot(Artboard.Name) // Create the root node with the name of the screen

IRTree ← HandleLayer(Artboard, currentNode) // Start building at the root

writeJson(IRTree)
Function HandleLayer(Artboard, currentNode):
    for layer in Artboard.Layers do
        class ← layer.Class
        switch class do
            case SymbolInstance do
                resultingSymbol ← handleSymbol(layer)
                currentNode.addChild(resultingSymbol)
                if layer has sublayers then
                    HandleLayer(resultingSymbol, layer)
                end
            end
            case Group do
                handleGroup(layer) // Groups, Columns and Decoupled Components
            end
            case Text do
                /* Text layers do not possess inner layers, so a simple add is enough */
                current.AddChild(newText(layer))
            end
            case Default do
                /* Other types of layer that we do not support */
            end
        end
    end
    return
```

A dot followed by a property name (for example **.Name**) relates to a specific property of a model that we are transforming. Equation 4.2 represents a transformation from the Name property in the Layer(Sketch) model into the DesignerName property in the Intermediate Representation model.

$$RuleExample1 : \frac{Layer.Name}{IRNode.DesignerName} \tag{4.2}$$

Some transformations can have the same Source and Target fields but can be done through different means. This is where the **(differentiator)** schematic comes in. The following equations 4.3 & 4.4 showcase this. Equation 4.3 showcases a transformation from a Layer to an IRNode through the X value, while equation 4.4 does the same but through the Y value.

$$RuleExample2 : \frac{Layer(X)}{IRNode} \tag{4.3}$$

$$RuleExample3 : \frac{Layer(Y)}{IRNode} \tag{4.4}$$

Every Layer is transformed into a IRNode based on their **symbolID** if they are of class "symbolInstance" (4.5), based on their **name** if they are "group" (4.6) or a simple conversion in the case of "text"(4.7).

$$Rule1 : \frac{Layer(symbolID)}{IRNode(symbolID)} \tag{4.5}$$

$$Rule2 : \frac{Layer(name)}{IRNode(name)} \tag{4.6}$$

$$Rule3 : \frac{Layer}{Text} \tag{4.7}$$

While this is true, what Rule1 does is that it fetches the Symbol's original Name in the template to be used as our "key" to identify the component. So in a way, I am always using the Name property as the "selector". The following flowcharts 4.12 & 4.13 showcase these transformation flows, which would correspond to algorithm 1 *handleGroup()* & *handleSymbol()*.

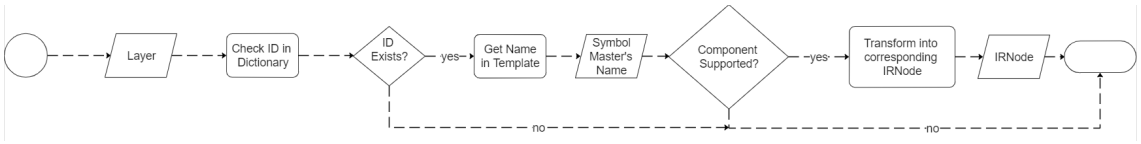


Figure 4.12: Class Diagram for the dictionary

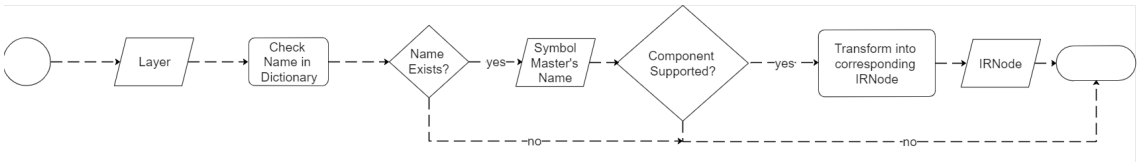


Figure 4.13: Class Diagram for the dictionary

Even though every component (excluding text since it does not need any identification) is identified by these 2 ways (by symbolID or by name), each of the 20 implemented components has their own transformation rules to match the component's requirements. Looking back at the class diagram for the Intermediate Representation represented in Figures 4.2 to 4.5, we can see the nodes posses a method called "Handle". This method is called after the creation of an IRNode and is in charge of complementing it with the necessary fields. In these next sections, I will present these transformations for the 20 components my work supports at the moment.

4.5.5.1 General Transformation

There are properties that are common through every component. These properties belong in the abstract class that can be seen in Figures 4.2 to 4.5. Most of these properties are set on the constructor since most of them can be gathered from properties of the source layer without difficulty.

The **DesignerName** property can be gathered from the *Name* property in the source Layer. It is preserved and used later to name the component in OutSystems

$$RuleDesignerName: \frac{Layer.Name}{IRNode.DesignerName} \tag{4.8}$$

The **InternalName** property is the component's "real name" i,e. is the name that the component possesses in the template. This value can be gathered from 1 of 2 ways depending if the Layer is a "symbolInstance" or a "group". These ways were already showcase in the flowcharts 4.12 & 4.13.

$$RuleInternalName: \frac{Dictionary.DictionaryItem.Name}{IRNode.InternalName} \tag{4.9}$$

The **RelativePosition** property, that contains the positioning of the Layer's position relative to its parent, can be easily gathered by converting the Layer's Frame property into it. The Frame property was already "cleaned" during the selecting attributes phase, and now it is a one to one transformation.

$$RuleRelatePosition: \frac{Layer.Frame}{IRNode.RelativePosition} \tag{4.10}$$

The **Type** property relates to the Class of the Layer. As a recap, here we store if it's an Instance, a group, text or the root node. The only transformation here, is that I used an Enum to store, so we preserve our 1 to 1 relation but instead of a string, it is stored as an int.

$$RuleType: \frac{Layer.Class}{IRNode.Type} \tag{4.11}$$

The **Children** property holds all the sub-nodes of a particular node. This property, initialized as empty, will grow as the sub-layers are transformed into IRNodes and added into here.

$$RuleChildren: \frac{Layer.Layers}{IRNode.Children} \tag{4.12}$$

The **Widget** property is a simplified name of the component, that helps my custom JSON converter to deserialize abstract classes later on during the second phase (Intermediate Representation $\longrightarrow$ OutSystems). This property is done hardcoded in the constructor in every component except the button.

The specific component attributes are all done in the Handle method after the component's instantiation. These next sections will describe their transformations per component.

4.5.5.2 Already Completed Widgets

Before going into specific Widgets, there are some that do not require any sort of extra transformations and are already complete after their instantiation into IRNode. To not create anything extra specifically for these components, they also call the Handle method after their transformation to IRNodes, but all it does is handle their Widget property. These components are: Dropdown, DropdownSearch, Form, Input, Pagination, Radio, Root, Search, Switch and Table.

4.5.5.3 Button

The button only possesses 2 extra properties that need to be handled: size and label. The label relates to the text that is presented inside a button, while the the size is, well, its size.

To gather the label, we have to look at its OverrideValues to see if any text override is present. If so, that represents the label 100% of the times. If not, a default "no text" string will be the label, as one is required when instantiating in OutSystems.

$$RuleButtonLabel : \frac{Layer.OverrideValues}{Button.Label} \tag{4.13}$$

For the size, we gather it from the SymbolMaster's Name that is now associated as the InternalName field of the button. Through string manipulation we can gather not only their size, but also what kind of button it is and we can already associate to the Widget field. For example, from `07. Widgets/01. Buttons/Primary Button/Default` we can gather its size is *Default* and it is a *Primary Button*.

$$RuleButtonSize : \frac{Button.InternalName}{Button.Size} \tag{4.14}$$

$$RuleButtonWidget : \frac{Button.InternalName}{Button.Widget} \tag{4.15}$$

4.5.5.4 Button With Icon

Similar to the Button, the Button with icon component also possesses a Label property. To gather its value, we conduct a similar rule presented at 4.13, changing the left-hand side component of the transformation.

$$RuleButtonLabel : \frac{Layer.OverrideValues}{ButtonWithIcon.Label} \tag{4.16}$$

This component differs from the previous for having an icon alongside the text in its interior. For this, it needed 2 more fields: a string containing the name of the icon (Icon) and a boolean to indicate if the icon is pointing to the right or on the left (only applicable to arrow icons).

For the alignment boolean, we can gather that information from the Node's InternalName property, if it contains Right or not.

$$RuleRight : \frac{IRNode.InternalName}{ButtonWithIcon.Right} \tag{4.17}$$

The Icon property follows the exact same procedure as the Label. It has a default value of a right or left arrow (dependent on the alignment property) and then it checks the original Layer's OverrideValues in search for an override of the Icon in the Button.

$$RuleButtonIcon : \frac{Layer.OverrideValues}{ButtonWithIcon.Label} \tag{4.18}$$

4.5.5.5 Card

The Card component is one of the most specialized ones. Like the previous component, this component also needs a boolean to indicate if the content is supposed to be displayed vertically or horizontally. This boolean can be acquired by checking the Node's InternalName, as in if it contains the "Vertical" or not.

$$RuleVertical : \frac{IRNode.InternalName}{Card.Vertical} \tag{4.19}$$

The Card is one of those elements that are solely created to group other components together. In this scenario, it's content has to be treated differently depending if it was implemented as a Group or as a SymbolInstance.

If it was a SymbolInstance, we need to give it its original internal components (i,e. the internal components in the SymbolMaster) and then search for overrides in the OverrideValue property of the original Layer.

$$RuleCardContent : \frac{Layer.OverrideValues}{Card.Content} \tag{4.20}$$

If it was a Group, this issue is resolved, and we simply transform it without this extra precaution.

4.5.5.6 Checkbox

This component searches the Layer's OverrideValues in search for overrides to either it's Label (the text to the right of a checkbox) or the checkmark's "box" (i,e. the box can be squared or circular).

$$RuleCheckmarkLabel : \frac{Layer.OverrideValues(string)}{Checkmark.Label} \tag{4.21}$$

$$RuleCheckmarkBox : \frac{Layer.OverrideValues(symbolID)}{Checkmark} \tag{4.22}$$

4.5.5.7 Columns

The Columns is the only OutSystems Widget in this project that is created by me to ensure structural integrity (later elaborated in section 4.5.9.2). This widget only contains 1 extra integer field to hold how many columns the component will be divided into.

Since the Widget was already created by me, this IRNode's name is already Column X where X represents this value, so with string manipulation we can grab it and store it more easily for the future.

$$RuleCheckmarkBox : \frac{IRNode.InternalName}{Column.NumberOfColumns} \tag{4.23}$$

4.5.5.8 Counter

The Counter widget is very similar to the Card in terms of its "handling".

It possesses a boolean called Vertical, to define if it is supposed to be oriented vertically or horizontally by default. To gather this value, a simple search for the word "vertical" in its InternalName suffices.

$$RuleCounterVertical : \frac{IRNode.InternalName}{Counter.Vertical} \tag{4.24}$$

Also like the Card, depending if it is instantiated as a Group or SymbolInstance, it will differ the way the component is handled. If it is a Group, then it will be treated as a group and its internal components will be treated as sub-nodes.

If it is a SymbolInstance however, it needs extra steps. Namely, we need to pick a copy of the original SymbolMaster and override its contents with the overrides that come in the Instance's OverrideValues property. In this case, we pick a copy of the Master because otherwise we would miss the non overridden components.

$$RuleCounteroverrideContent : \frac{Layer.OverrideValues}{Counter.overrideContent} \tag{4.25}$$

4.5.5.9 Group

The Group widget only has 1 property that is handled in the Handle method. This property is Flex which is a boolean that represents if the group is a Flex Group or a simple Group. Flex Groups are normal Groups but that will, in the future, possess the CSS attribute "display:flex" to place their contents horizontally instead of vertically stacked.

Similarly to the Columns, since Flex Groups are created by me (later elaborated in section 4.5.9.2) and I always name their InternalName as as "Group Flex", it is easy to verify if they are Flex Groups or not.

$$RuleGroupFlex : \frac{IRNode.InternalName}{Group.Flex} \tag{4.26}$$

4.5.5.10 Icon

The Icon widget possesses 1 extra info: the Icon name. This is a string that contains the Icon's name that is present in the template that matches 1 to 1 with the ones in the OutSystems library.

However, the name in the template is filled with filler that is there for organizational concerns but that we do not need, as can be seen in Figure 4.14.

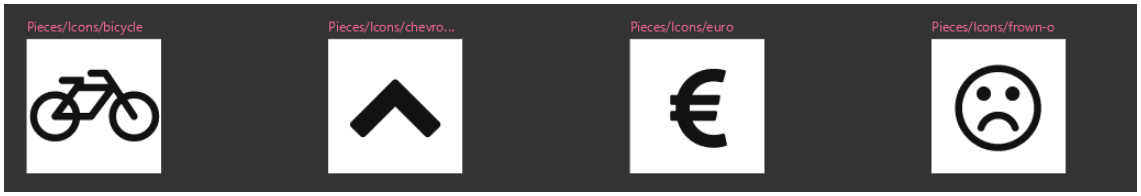


Figure 4.14: Snippet of Icons in the OutSystems UI template

By observing the image, we can see that these organizational parts are always prefixed and that the icon's "real" name is always after the final and that is what we save.

$$RuleIconIconName: \frac{IRNode.InternalName}{Icon.IconName} \tag{4.27}$$

4.5.5.11 Text

For the Text widget, we save 2 extra properties: a string containing the correspondent text to be displayed and a double containing the FontSize.

To get the text's string we can look at the Layer's attributedString._string value.

$$RuleTextString: \frac{Layer.attributedString.\_string}{Text.String} \tag{4.28}$$

To gather the FontSize we need to check the array of attributes that attributedString possesses. The position we need to check is always the first one as it is the one that relays the attributes of the font.

$$RuleTextFontSize: \frac{Layer.attributedString.attributes[0].attributes.msa.attributes.size}{Text.FontSize} \tag{4.29}$$

4.5.5.12 UserAvatar

The UserAvatar widget can be separated into 2 widets: an image placeholder and a text that is usally the person's name. Since the first component is an image, we ignore it as later it will be replaced automatically with the initials of the name (E.g. SR for Sofia Ribeiro).

For the person's name however, the tool does support it's instantiation. This component differs from the rest as what we normally would do (search for override values) does

not apply. Here, the text belongs on a different sub-layer, and so we have to go search for it and attribute it to the UserName property. I do this so it is associated with the UserAvatar component and it is instantiated as a whole component.

$$RuleUserAvatarUserName : \frac{Layer.Layers(text)}{UserAvatar.UserName} \tag{4.30}$$

4.5.6 Variations

Most Symbol Instances are not created as an exact copy of the Master. Designers usually change a few parameters, such as colors or the presented text on a button. While I could implement the Master and leave it be, I decided to implement these variations. These variations are stored in the **overrideValues** property of the Symbol Instance, which is an array of objects and can be seen in Fig 4.15. These objects contain two pairs of key values:

- **OverrideName:** A string composed of 2 major parts. Before the underscore is the ID of the Layer to be overwritten and the after part is the type of overwriting. Looking at the provided figure, the first object is referring to an overwrite of a symbol (changing symbol X with symbol Y) while the other 2 are text transformations.
- **Value:** The value to be overridden by. In the case of a text transformation, it represents the new string. In the case of a Symbol transformation, it represents the new Symbol's ID.


```
"OverrideValues": [
  {
    "OverrideName": "DCFAC4AE-D7C1-4417-BC25-99CF17C91B03_symbolID",
    "Value": "103FB8F5-47CD-4F5C-9448-A10E4E838EF6"
  },
  {
    "OverrideName": "195EACA7-9EF5-47CB-A5E3-54BBEC33D701_stringValue",
    "Value": "89"
  },
  {
    "OverrideName": "C588BFC9-E2E7-438F-803F-8A9FE7537316_stringValue",
    "Value": "Pending\ntasks for today"
  }
]
```


Figure 4.15: Example of the OverrideValues property

4.5.7 Nested Layers

Some UI components have other nested components that can be either required or optional. An example of a required nesting is the Card component, whose purpose is to layout its inner components. By itself, it does not possess any purpose. An optional nesting could be an Icon inside a Button component, where it can be added to present the user with a more straightforward interpretation, but without it, the Button can still fulfill its purpose.

The concern of representing this nesting on my Intermediate Representation is trivial as, just like in the Layers, this is represented by the descendants in the IRNode tree that it creates.

4.5.8 Vertical Placement

When working with the Sketch Model, although the Layers possess the X & Y coordinates in their *Frame* property, the Layers themselves are not ordered in the JSON files. The tree of IRNode I am creating should already be ordered by the Y coordinate in order to facilitate the instantiation later, as the construction in OutSystems is done from top to bottom. This vertical alignment was easily accomplished by ordering the *Layer.Layers* in the *HandleLayer* in the presented Algorithm 1 function before iterating through them.

4.5.9 Horizontal Placement

Horizontal placement however, was a bigger challenge. While at the start all I was doing was creating a stack of components by instantiating them on top of each other by their Y axis value, now I needed to do them side-by-side. This meant, I need to check if any layers overlapped and then group them together somehow.

4.5.9.1 Overlapping Layers

Before starting to brainstorm ideas on how to create the algorithm to calculate the overlaps and define the whole page layout, I decided to ask professors and in communities if they knew any known algorithm that already does what I require, to not waste time reinventing the wheel. Both sources came with the same response: they were not aware of its existence.

The algorithm I developed has two major steps and it is executed before entering the *for* loop seen in algorithm 1. This means that this algorithm will be applied on the sub-Layers of the passed Layer parameter, getting them "ready" for the following loop. The first step is called "tagging", where the purpose is to identify for each component who they overlap with. Figure 4.16 showcases how the nodes would be represented after this "tagging" process.

The second step is the "grouping" step. The idea behind this step is to create a new group to hold all of the horizontally overlapped components so they can be prepared and more easily identifiable for the instantiation in OutSystems. The idea is to remove the overlapping Layers from their original parent Layer, create a new grouping Node, adding the overlapping Layers to the newly created group. The final step is to add this group as a Sub-layer of the original parent.

Looking at the example in Figure 4.16, *A*, *B*, *C*, *D* and *E* should be together in a group, as these overlap with at least 1 other node. So these nodes would be taken out of the *root*'s sub-nodes, and added as the sub-nodes from a newly created group node. This new

```
{
  "id": "root",
  "children": [
    {
      "id": "A",
      "overlaps": "C"
    },
    {
      "id": "B",
      "overlaps": "C"
    },
    {
      "id": "C",
      "overlaps": "A, B, D
    },
    {
      "id": "D",
      "overlaps": "C, E"
    },
    {
      "id": "E",
      "overlaps": "D"
    },

    {
      "id": "F",
      "overlaps": null
    }
  ]
}
```

Figure 4.16: Example tagging

group node would be added back as a sub-node of *root*. This results in a structure seen in Figure 4.17.

4.5.9.2 Columns vs Flex

When creating the Node to group these overlapped layers together we considered two options: I could use Columns or a flex group. To understand when to use which, I went to past OutSystems projects and tried to come up heuristics. These ended up being the composition of the "groupings". For example, a row of cards would be used with Columns, while text with an input and a button a simple group would suffice.

The Columns widget has the advantage of automatically handling the behavior of each of its columns whenever an event requires it (for example when a page/screen is resized) at the cost of overall performance.

The Flex Group option relies on grouping the overlaps in a single container with the inline style of "group:flex". This property allows for its contents to be placed horizontally.

```
{
  "id": "root",
  "children": [
    {
      "id": "Group",
      "children": [
        {
          "id": "A",
          "overlaps": "C"
        },
        {
          "id": "B",
          "overlaps": "C"
        },
        {
          "id": "C",
          "overlaps": "A, B, D"
        },
        {
          "id": "D",
          "overlaps": "C, E"
        },
        {
          "id": "E",
          "overlaps": "D"
        }
      ]
    },
    {
      "id": "F",
      "overlaps": null
    }
  ]
}
```

Figure 4.17: Example of the grouping step

This creates a better performance than the columns widget, but its behavior has to be manually created by a developer. Not only that, but this approach creates inline styling, which goes against a front-end practice the developers at OutSystems follow. So while support for this approach exists, it is currently disabled.

4.6 Intermediate Representation To OutSystems

The second step of the implementation is the transformation from our Intermediate Representation model to the OutSystems one. At the end of the previous step, we have one JSON file per application screen. We will take advantage of OutSystems' ModelAPI. This API allows us to programmatically handle OutSystem's model to create a multitude

of things such as applications, screens, components, styling, and so on. For this project, we use it to programmatically create the screens and their respective components. Besides our JSON files, we will also be receiving an OML file containing the already instantiated LSG UI components. With this file, the ModelAPI allows us to use it in two ways: either use it as a template and create our own as a replica or edit the one given.

A particularity that must be discussed about these OML files that impacts how I approach them is how they are internally structured. These files have keys that are their unique identifiers and are generated when they are created. Since these function as identifiers, these keys are essential for the inner workings of OutSystems' applications in order for them to work correctly. Using the file as a template and creating a replica, these keys are getting re-generated by us on runtime, and so our created OML file will have different ones than the OML provided. Since I do not want to cause any disturbances that would lead to extra work for the development team, I will be editing the provided OML file instead, injecting in it my generated screens. This will ensure that the keys remain the same, and so any internal logic that requires them will continue to work as intended.

4.6.1 JSON Treatment

Just like the first step of our process, this one will also start by manipulating JSON files. While on the other one, I had to create our own class for JSON.NET deserializer to convert the JSON objects into, here I had already created this element in the previous step: the IRNode. A problem with this method appeared as I am now passing an **abstract class** to the deserializer. This meant that when looking at a node, the deserializer could not convert it automatically due to ambiguity issues.

To solve this issue, I implemented a custom converter that is passed to the deserializer to help solve these ambiguities. This converter looks at the Widget field in the JSON objects, and based on it; it creates the appropriate IRNode. For instance, an object with Widget Checkbox would deserialize to a Checkbox type IRNode. With this new converter applied to the deserializer, JSON.NET will create a tree of IRNodes per JSON file facilitating the process.

4.6.2 Component Instantiation

Having the JSON deserialized in a tree structure of IRNodes, the instantiation of the components revolved around doing a depth-first traverse through it. Each tree represents a screen in the application, and so it creates a new screen with the same name as that was in the designs, which is stored in the Root node. OutSystems requires every screen to be a part of a ScreenFlow, and so I create one named *GeneratedScreens* where I will insert every screen the program makes.

Every screen in the OutSystems model has a Layout. These Layouts have different placeholders to help the users place their components in certain places. Examples of these placeholders are the MainContent (the center body of the page) and the footer (the

bottom of the page). Here, the program will place every component in the MainContent as these Layouts can have different placeholders, but the MainContent placeholder is constant.

Algorithm 2 showcases this process.

Algorithm 2 Instantiation of the IRNodes to OutSystems

```
Data: A tree structure of IRNodes
for screen in JSONfiles // Do for each of the of the jsons
 do
    treeOfNodes ← Deserialize(screenX.json)
    /* create a screen using the root's name */
    screen ← createScreen(listOfNodes.root.name)
    /* fetch the reference of where to place our components */
    placeholder ← screen.getMainContent()
    for node in treeOfNodes.root // start iterating through the nodes
     do
        /* Every element must be enclosed in a container */
        container ←new Container(placeholder)
        node.InstantiateWithContainer(container, node.Children)
    end
end
/* Every IRNode has this method adapted to its needs */
Function InstantiateWithContainer(container, children):
    instantiateComponent(container)
    if component has children then
        for node in children do
            /* check if the node has placeholders to put content */
            placeholder ← node.getPlaceholder()
            if placeholder not null then
                node.InstantiateWithPlaceholder(placeholder,node.children)
            else
                childContainer ← newContainer(container)
                node.InstantiatewithContainer(childContainer,)
            end
        end
    end
```

To instantiate a component using the ModelAPI there are two major approaches. The first one is using the native support that the API has for some components. For these components, their instantiation is trivial and can be done with one single line of code that follows the following structure.

```
var component = container.CreateWidget<Component Interface>()
```

This approach is restricted to only a few components that are a part of the OutSystems library. Most of the rest have to be instantiated by reference.

To do so, the program generates a generic WebBlock and assigns that WebBlock's **source block** as the correct reference of the desired component. The provided OML file

already contains the required references to the complete OutSystems library, and through that, we can get any UI component we require.

At the end of this process, we save the file with the structured application pages and a report file is output in the same directory.

The output report file mentions which components were not recognized and which were but are not supported yet. This identification is done by the Layer's name and position in the designs so Developers can easily search for them.

At this point my tool has finished the whole process and a summary of it can be seen in Figure 4.18.

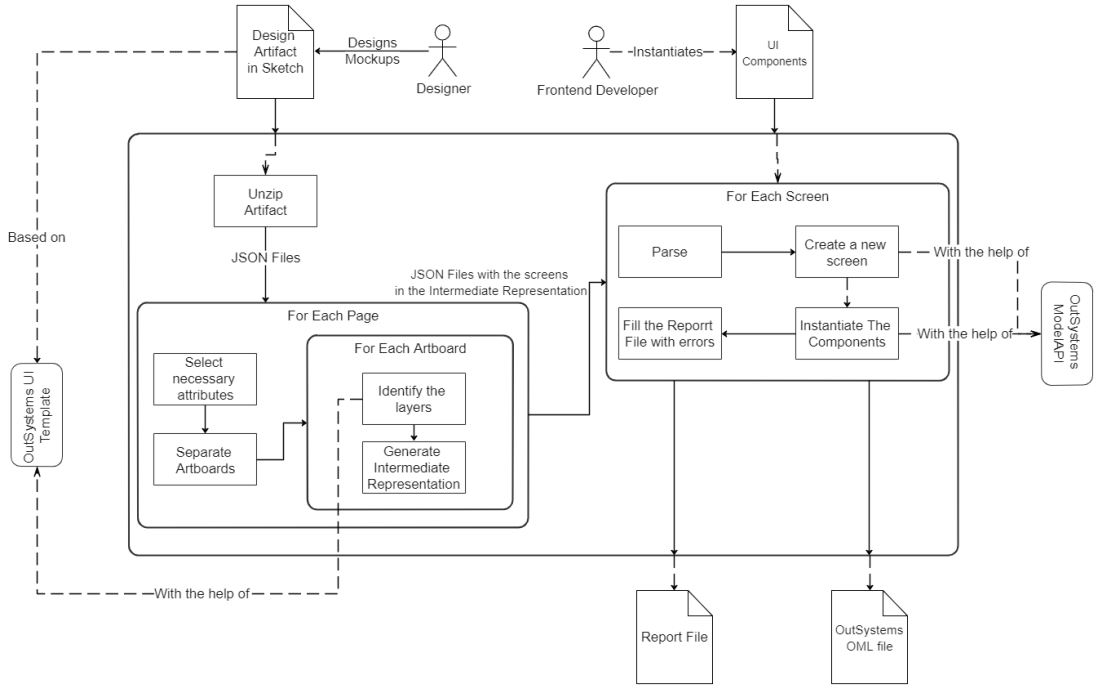


Figure 4.18: Summary of the process.

4.6.3 Sub-Goal Validation

When implementing support for the first set of the 20 components I also verified the possibility of doing the secondary goal: instantiating part of the components, such as assigning them necessary auxiliary variables. I decided to drop this goal and focus solely on the primary as this one presented a bad return of investment. Although the ModelAPI allows to instantiate the necessary variables and structures and assign them, I cannot detect any interaction between components from the designs, so the concept of shared variables disappears. This would lead me to a possible solution: creating a new variable per component that requires it. Doing it this way would most certainly lead to excessive variables per page, which would fall on the development team to clean the extra and

assign the correct shared ones. This goes against my Precision focus, and so I decided to drop this goal altogether.

4.7 Chapter Summary

In this chapter we began with a high-level view of the solution and discussed important topics as we zoomed in on our approach. It went through all of the functional increments that we took to implement our tool and most importantly, the decisions that we had to take. These decisions were important to decide (what we thought were) the correct approaches to lead the work, such as to focus on "precision" over "fullness" by only instantiating what we are close to 100% certain that is correct, or to hard-coded the transformation rules instead of using an MDL (full discussion of this topic on section 6.1).

A key point of this chapter was the Intermediate Represent and how I created it to have a technology in the middle of the transformation independent of both models for an higher abstraction and adaptability to different models from the ones we used.

Finally, the other major key of this chapter were the challenges that are present in a work like this. Mainly, the challenges that are related to the phase that handles the design side of the work. This "first phase" out of the two main phases that divide the tool(design to intermediate, intermediate to front-end) is subjected to many variables that condition our implementation since we have to account for them but also know when its "enough". These are mostly related to the heterogeneity of the design practice (how a designer can build the designs in innumerous different ways).

5

Results

To evaluate the implementation, I measured the Precision and Recall statistics in 6 real past projects from OutSystems using their formulas (5.1, 5.2) respectively. Although these metrics are mostly used on classification problems, I adapted their concepts into our work. Precision relates to the tool's ability to identify a component correctly and Recall relates to the percentage of components the tool is able to identify in the designs.

$$Precision = \frac{truePositive}{truePositive + falsePositive} \tag{5.1}$$

$$Recall = \frac{truePositive}{truePositive + falseNegative} \tag{5.2}$$

To calculate the total amount of components in each project, our tool already passes through every layer in the designs, so it was easily calculated. To count the True Positives, False Positives and False Negatives (True Negatives do not exist in this type of work) that are necessary to calculate the Precision and Recall statistics, these had to be manually counted. In our case, True Positives relate to how many components we are correctly identifying, False Positives are all the components we are identifying incorrectly and the False Negatives are all the other components that are calculated by subtracting the previous 2 from the total amount of components, since in our case True Negatives do not exist.

Table 5.1 summarizes our results. We achieved a global precision of 99.6% and recall of 80%. So, developers get a low number of misclassified components, mitigating the potential rework effort fixing wrongly create components, and the tool creates around 80% of the components, which is rather good, considering it only uses the implemented transformations for **20** out of the 87 OutSystems components.

At OutSystems there is a Customer Success team of experts composed of, at the time of writing, 5 Front-End experts. This team is responsible for hand-crafting the Live Style Guide by translating the design screens and components into the OutSystems language. We sent this team our tool and the same six past projects to apply our tool and check the results by inspecting every single generated screen to compare the results of their work with the results obtained as the output of our tool. Since they are experts in this area and we are automating part of what they do, their feedback is the most important.

Projects	Comp	TP	FP	FN	Precision	Recall
A	3310	2237	18	957	0.992	0.709
B	5876	4079	16	1057	0.997	0.820
C	1943	1378	2	382	0.999	0.803
D	4178	2994	17	713	0.995	0.829
E	569	434	5	102	0.989	0.819
F	1981	1634	5	266	0.997	0.865
Summary	17857	14317	63	3477	0.996	0.805

Table 5.1: Precision and recall for our 6 projects

Dev ID	PU (out of 7)	PEOU (out of 7)	Mental	Physical	Temporal	Performance	Effort	Frustration	Raw TLX
1	4	6	30	10	40	30	60	100	35
2	3	6	20	20	20	20	10	10	18.33

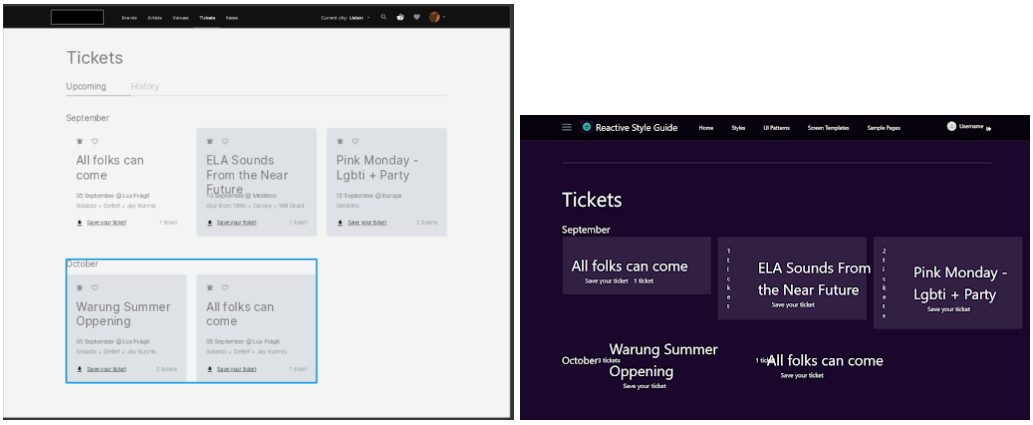
Table 5.2: TAM & NASA-TLX results

These past projects were created without having our tool in mind and so results will reveal a significant variance. Alongside this package, I also sent a survey for the team to fill. This survey had questions related to each project and some more general questions based on TAM [47] and the NASA-TLX [61] frameworks. We got feedback from 2 of the 5 developers.

Table 5.2 shows the results gathered from the survey regarding the TAM and NASA-TLX frameworks. Although these tests were created to be applied on an bigger sample, we got results from 2 out of the 5 candidate surveyees and decided to include, even if just for statistical reasons. Through them, I found that the tool's perceived ease of use (PEOU) is high (6 out 7), while its perceived usefulness (PU) has an average of 4 out 7. In terms of the overall workload, all of the NASA-TLX factors are low, except for effort and frustration, which is expected due to the less effective projects causing some extra effort. Nevertheless, they commented that after some more development and minor improvements, it could very well be a staple in their future workflow. These improvements were mostly focusing on styling issues (such as identifying styling classes and applying them to the components) which fall out of scope of this work.

In terms of the 6 projects, the tool's effectiveness was worse when applied to designs where it was not only more challenging to identify the UI components correctly but also in those with a higher level of complexity. This complexity could be due to several reasons such as a higher presence of custom patterns or complex styling related techniques which we cannot detect. An example of such case can be seen in Figure 5.1.

There are also theoretically, worst-case scenario projects where Developers would not use the result created by the tool as nothing created could be used. Theoretically since



(a) Screen Mockup (b) Generated Screen

Figure 5.1: Example of not so successful generation

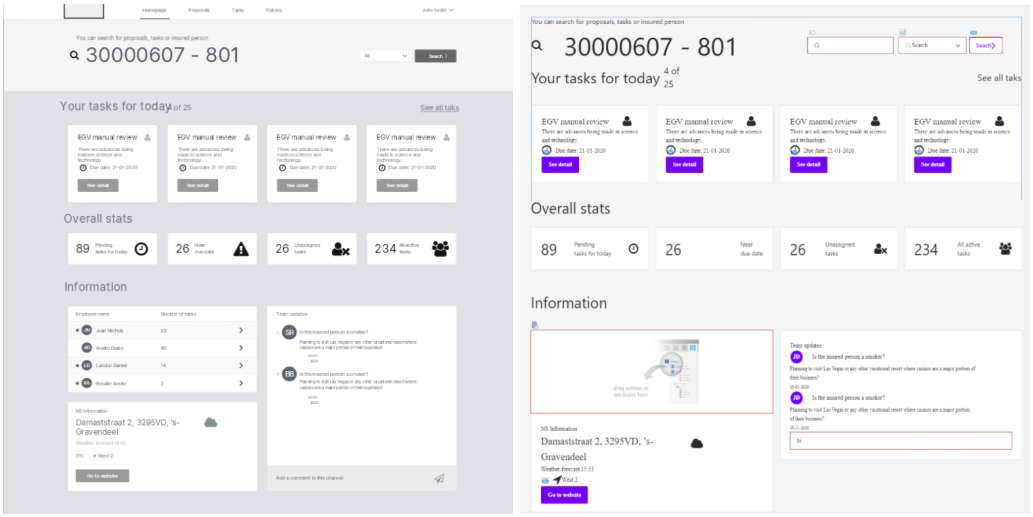
we did not observe this occurrence in any of the projects. In these cases, developers can quickly delete any generated screen by deleting the created Screen Flow, or simply ignore it and use the fresh file that was used as input causing no significant time lost.

For projects where our tool was more effective, the Front-End team can save up to 8h of the 16h they have (see table 5.3, project B). This is a 50% time reduction of the two workdays they usually have to invest in creating these pages. Developers noted that they expect they could deliver up to 400% more pages if they started working on top of the output, as can be seen in projects B and C, where they could deliver 6 more pages on top of the 2 they originally delivered. An example of a screen generated from one of these projects (project B) can be seen in Figure 5.2.

Projects	Time to create the set	Extra Pages (delivered)
A	28h	1 (2)
B	8h	6 (2)
C	16h	6 (2)
D	20h	0 (3)
E	10h	2 (3)
F	24h	0 (2)

Table 5.3: Developer Feedback for the 6 projects

The results reflected somewhat what I already expected. Even on projects that take the same or longer time to recreate the same pages (such as projects A and C) that were delivered at the end of the service, the development team still looked at the other pages that we generated automatically and found some pages that they could work on and at the end deliver the double or more pages than they initially did.



(a) Screen Mockup (b) Generated Screen

Figure 5.2: Example of a successful screen generation

6 Discussion

Since this transformation process is divided into two main phases, this chapter will be divided into two main sections corresponding to each of the phases to better group the discussions with the appropriate phase and a final section discussing some of the results.

6.1 Transformation Process

Before going into a more in-depth discussion related to each of the two phases, first we need to discuss something that covers both of them: how we implemented the transformations. We could have used model transformation language such as ATL or QVT, but ultimately decided to work with the internal representation of the files in JSON, and hard code our transformation rules. The focus of this work, was to validate our hypothesis and answer the question: *can we improve the design-development collaboration process, by partly automating the generation of screen through the composition of reusable UI components with design mockups?* This validation meant, not only the implementation of the tool, but to also test it with domain-experts (the potential future users of our work) and gather as much of their feedback as possible and also gather feedback from this work's group of stakeholders. With all of this in mind, plus the time-restraint that comes coupled with the work being developed as a Master's thesis, we opted to be pragmatic and took these decisions related to the transformation process.

6.2 Design to Intermediate Representation Phase

One of the most complex challenges in this phase is one that is embedded in this sort of work: the subjectivity associated with the design process. Different designers do their work differently, or even the same designer can do things differently in different works or even in the same work. This heterogeneity leads the designs to be structurally different in their internal representation, which makes automatic detection more difficult and less accurate.

6.2.1 5 Guidelines

I created five simple guidelines that mitigate the top challenges that affected this work to combat this somewhat. These were created with the idea of being as unintrusive as possible while still be highly effective, so they are more readily accepted into the design process.

These were as follows:

- Group layers as much as possible.
- Name the design page as "UX Design"or "UI Design".
- Leave the design as clean as possible.
- Try to use the Symbols as much as possible.
- If a component made manually matches/is supposed to be an LSG component, like a Card, name it the same as it is in the Symbols.

These next subsections will discuss these guidelines, their reasonings and the impact they can have in creating designs more "ready"for approaches that rely on strategies similar to the ones I used.

6.2.1.1 Guideline 1 - Grouping

Design tools such as Sketch allow for designers to group their layers in a group component. This can be helpful to the designers while they are designing the mockups, it is even more critical for processes like mine that focus on accurately identifying components through their internal structure. While to a human looking at the designs, it is natural to create a grouping association between elements, automatically, this concept is lost unless it is explicitly created. Observing Figure 6.1, the highlighted area comes to us naturally as a group, but observing with attention, one can see these components are not “grouped” in the design tool, so that this concept will be lost.

Another thing that grouping affects is the concept of columns. By observing in Figure 6.2, looking at either the row of cards or the row of counters below, again naturally to an expert developer and us, it is deduced we have four columns, one per element. This comes naturally to us since we see four different grouping components side to side. If this concept of grouping is lost, i.e., the counters were not grouped, the algorithm would pick up the 12 elements (one number, one text, and one icon per counter) in a row and deduce it needs 12 columns and will create an unexpected result.

6.2.1.2 Guideline 2 - Naming the Design Page

This one is relatively simple but can save approaches like this time and computational power. Again, since the design process is very subjective, designers can call the Page to

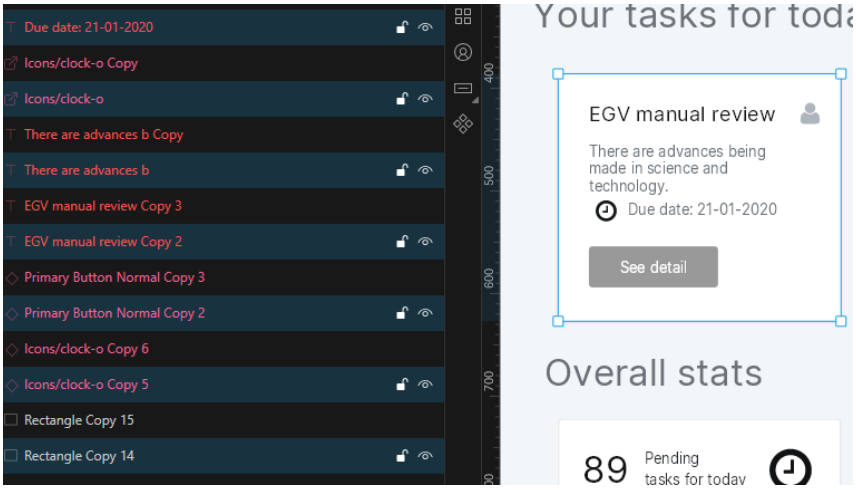


Figure 6.1: Example of a group not explicitly grouped.

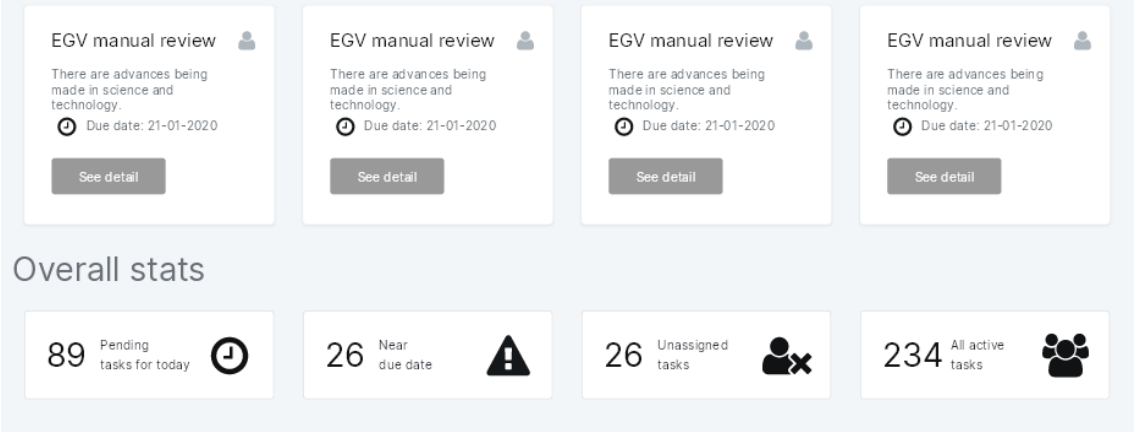


Figure 6.2: Cards and Counters follow a 4 column structure.

create their designs whatever they want. This causes for works like this is the need to figure out which Page contains the screen mockups. Observing Figure 6.3, one cannot figure out which is the correct Page where the UX or UI design mockups are located.

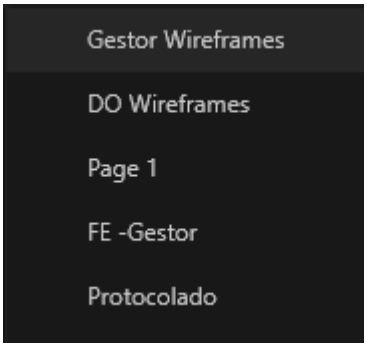


Figure 6.3: Example of difficult to find designs.

In order to figure out which Page is the correct one with reliable precision, the program would need to inspect every JSON file thoroughly and try to identify the mockups

through some parameters. This guideline proposes to mark these screen mockup Pages with a constant name such as "UX Design"or "UI Design,"which facilitates the correct identification.

6.2.1.3 Guideline 3 - Cleaning the Design

During the design's creation, designers usually experiment with different components to see the appropriate component to place. What sometimes ends up happening is that designers place a component over another, completely covering it. While for them and the developer that receives the designs, since the component is hidden, it is the same as if it did not exist, the fact is that it remains present in the design and, more importantly, in the internal representation, which works like mine will pickup and instantiate is a standard component. This guideline proposes that designers try to leave the designs with only the visible components. Observing Fig 6.4, the highlighted text "Take control with a" is hidden behind the Card. Visible or not, the tool will pick the layer up and instantiate it like normal.

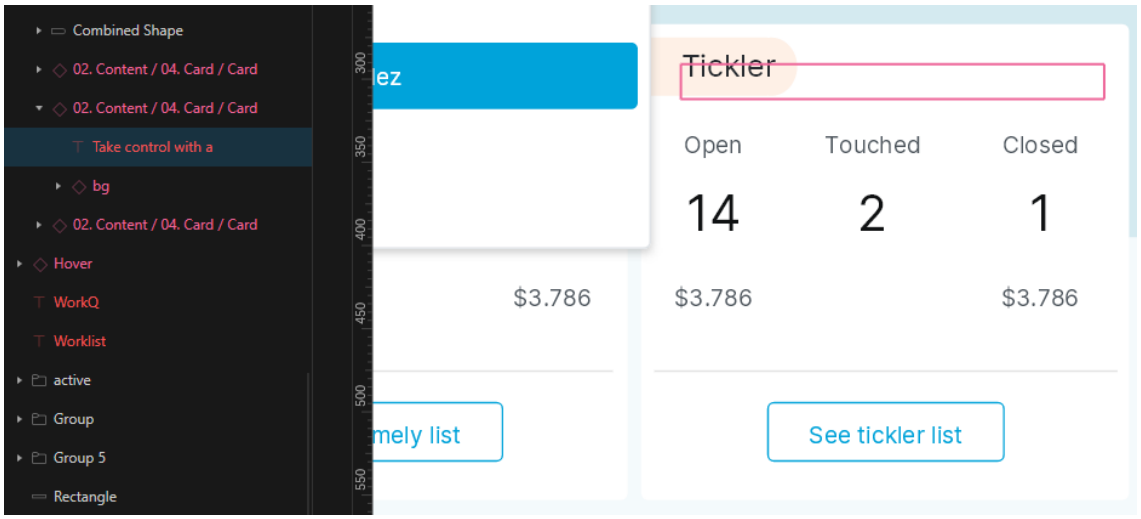


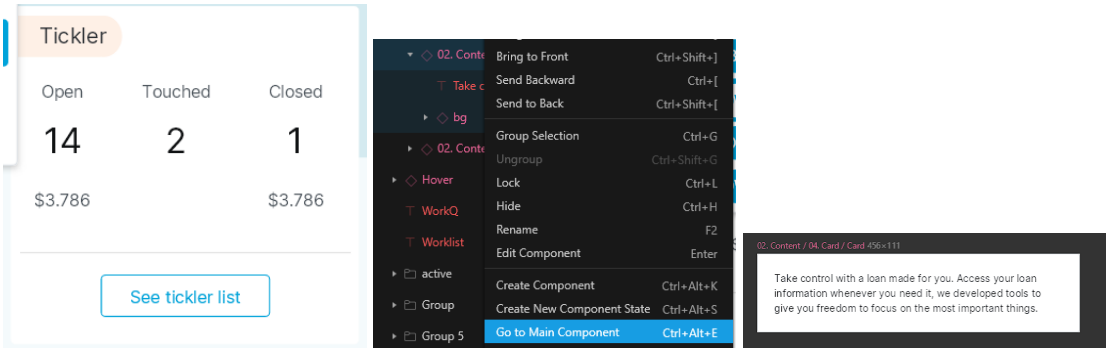
Figure 6.4: Hidden text Layer under a group Layer.

6.2.1.4 Guideline 4 - Use the Symbols

The fourth guideline relates to the use of Symbols or their counterpart in other design tools. Namely, to use Symbols as much as possible during the design process, not only for the benefits these naturally bring but also to keep the components' relation with the component in the templates, making identification somewhat trivial in tools like mine. Figure 6.5 showcases this relation between an Instance and a Master.

6.2.1.5 Guideline 5 - Name Components created from scratch the same as in the LSG

A point that we identified during the research was that the use of Symbols is not always the best for the designers. When using them, designers are very restricted in terms of



(a) Layer Instance in a Design (b) Layer Reference (c) Layer Master in Template

Figure 6.5: Interconnection of Layer Instances and Master.

variations they can apply to Symbol Instances. While a simple text or icon swap is easily done, more complex variations such as embedding another Symbol in an Instance are impossible. Another reason designers choose to stay away from Symbols is time. Some components can easily be remade from scratch, rather than a few clicks and searching for the correct Symbol. An example was already provided in the thesis, but to reiterate: a Card which is a group with a rectangular white background, can be easily recreated and recreating it is faster than looking for the Symbol.

These reasons often lead designers to create the components from scratch instead of using the Symbols. This guideline asks for the designer, when creating the layer from scratch, to name it the same as it is in the template, as can be seen in Fig 6.6. This way, we have a connection to components in the template, and we can more accurately identify the component.

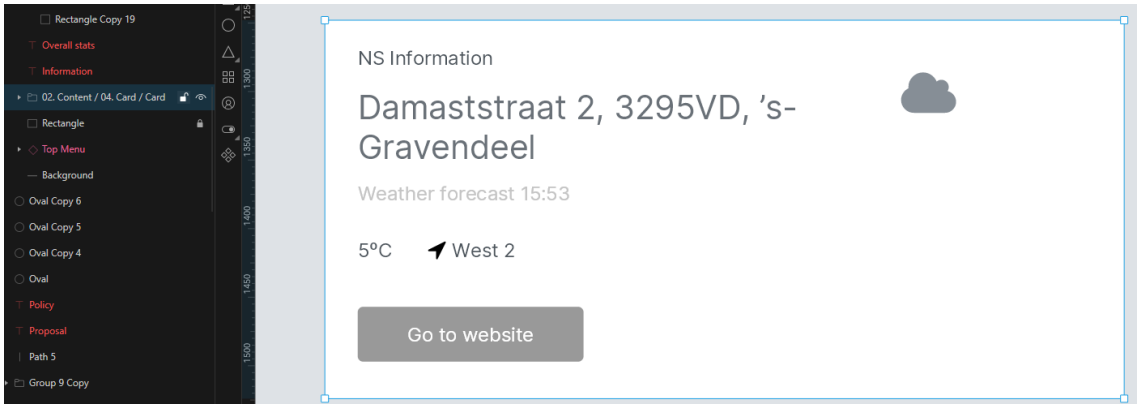


Figure 6.6: Card component created from scratch named after the Symbol Master

6.2.1.6 Designer Guideline Feedback

These five guidelines were discussed with the Design Lead over at OutSystems. The biggest challenge the design team faces is time and so inserting more guidelines in their work would prove difficult. However, it was said by the Lead that these guidelines were

unintrusive and could be very well incorporated into their work when developing designs to use in works similar to this.

6.2.2 Template Versioning

Another concern that appears in the design process is the number of existing versions of the template.. Different designers were shown to have different versions of the template, some more updated than others. My work is loosely coupled with the version of the template I used since, as seen in section 4.5.3.1, the template's only purpose is to create an auxiliary structure built on runtime to help identify the components in the mockups. The only way this different versioning would negatively impact this work is if there would be a radical change from one version to another, which would require some tweaking but not much.

6.3 Intermediate Representation to OutSystems

There is not much to be said or discussed about this phase. Some parts of this phase's implementation, were reused from the first phase, since these were already readily setup. Namely the process of handling the deserialization of the JSON files. One particularity however, of this deserialization, was the need to create a custom converter to help JSON.NET deserialize our abstract IRNodes into their respective component's class.

This phase had by far less challenges, as most of them are related to the design, due to the heterogeneity of the design process. Here, we were working with our own Intermediate Representation as input, and these already went through the Design-related challenges making this phase far easier. The few challenges that came during this phase were mostly due to how recent OutSystem's ModelAPI is. Since there was no official documentation for the API at the time of working on this thesis, a significant time was dedicated to trial and error testing and debugging alongside some questioning made to the OutSystems experts in charge of the ModelAPI.

7

Future Work

When developing this work, several topics worth of being investigating arose.

The first of these would be to try and diminish some of the set restrictions we had to do. Namely, it could be interesting to develop a way to diminish the required guidelines we presented in the discussion chapter at 6.2.1. Even considering I set up those guidelines as not intrusive as I could and, even after having discussions with the Design lead, it still presents an excellent opportunity to evolve this work to cut some of those necessities off or even all of them.

A possible future work appeared when investigating the current development workflow that was discussed in section 2.1.6.2. While this work accelerated their first step, we deemed the second step as a not good investment since the interactions between components are not present in the designs. It could be interesting to investigate an approach that tries to infer these interactions. Besides these two, there remains the third step which I did not approach with my work: bootstrapping the logic to populate the page with sample data. During our interviews, this process was identified as a time-consuming process in the development phase, so accelerating it would be a great opportunity.

Another possibility is the option mentioned above to adapt this work to other design and front-end technologies. Here we instantiated the hypothesis and applied it to the Sketch design tool and the OutSystems low-code technology, but the work's core is not restricted to this pair. It would be interesting to see it adapted to other technologies such as Figma, another of the top Design Tools, and some other low-code technology.

Finally, it would be interesting to pick up our presented approaches and try to develop a hybrid approach leveraging the tool users' expertise to accomplish more complete pages while preserving high accuracy. Here I focused on preserving accuracy rather than completeness, which left pages with missing components. Implementing a hybrid approach that focuses on Precision and Recall could produce even better results. Whenever the program would find a component where it would not be sure to instantiate or not, it could ask for the user's input and continue to do so at eternum or, after some interactions, it could train and evolve to make these decisions by itself when it hit a threshold of certainty.

8

Conclusion

Our goal was to mitigate an existing gap in the collaboration between designers and front-end developers. This is vital for a great User Experience, which is a crucial factor in today's market and to cope with the current scarcity of available individuals with this expertise in the market. We implemented a solution that partially automates the generation of structured application screens in a LowCode technology - work done manually today - by taking advantage of a design technology's structure and a library of LowCode reusable UI components.

The results gathered from an OutSystems professional front-end development team suggest our tool may improve the value these teams can deliver to their customers. Depending on the complexity of the projects that the tool is being applied to, the team reported increases between 150%-400% in application screens created with a similar effort. In addition, by automating part of the composition process, we freed up time for these front-end professionals that they can redirect to other topics. In the worst case scenario, where none of the generated screens are useful, the cost of applying the tool is nearly negligible. The developer can always choose to ignore the generated output and start the screens from scratch, as they normally would, costing them a couple of minutes assuming they were not working in parallel or running it in the background.

Our approach was based on unidirectional model transformations using an intermediate representation between our source and target models, which created a broader abstraction scope and allowed for an easier way to adapt to technologies different from those we used. Not only that, but the fact that this representation is a tree structure built by abstract nodes makes the tool easier to be evolved and extended to support more UI components.

The biggest challenge we encountered when implementing our solution was the heterogeneity of how UX and UI designers work. There are differences from one designer to another, but there are also notable inconsistencies in the approaches followed by a single designer for a particular project. These have severe implications on any approach that follows a similar path to mine as it impacts the tool's ability to accurately identify the used UI components, thus impacting the portion of designs the tool can convert automatically.

When developing this work, several topics worth of being investigating arose. The first of these would be to try and diminish some of the set restrictions we had to do. Namely, it could be interesting to develop a way to diminish the required guidelines we presented in the discussion chapter at 6.2.1. Even considering I set up those guidelines as not intrusive as I could and, even after having discussions with the Head of Design Practice, it still presents an excellent opportunity to evolve this work to cut some of those necessities off or even all of them.

A possible future work appeared when investigating the current development workflow that was discussed in section 2.1.6.2. While this work accelerated their first step, we deemed the second step as a not good investment since the interactions between components are not present in the designs. It could be interesting to investigate an approach that tries to infer these interactions. Besides these two, there remains the third step which I did not approach with my work: bootstrapping the logic to populate the page with sample data. During our interviews, this process was identified as a time-consuming process in the development phase, so accelerating it would be a great opportunity.

Another possibility is the option mentioned above to adapt this work to other design and front-end technologies. Here we instantiated the hypothesis and applied it to the Sketch design tool and the OutSystems Low-Code technology, but the work's core is not restricted to this pair. It would be interesting to see it adapted to other technologies such as Figma, another of the top Design Tools, and some other Low-Code technology.

Finally, it would be interesting to pick up our presented approaches and try to develop a hybrid approach leveraging the tool users' expertise to accomplish more complete pages while preserving high accuracy. Here I focused on preserving accuracy rather than completeness, which left pages with missing components. Implementing a hybrid approach that focuses on Precision and Recall could produce even better results. Whenever the program would find a component where it would not be sure to instantiate or not, it could ask for the user's input and continue to do so at eternum or, after some interactions, it could train and evolve to make these decisions by itself when it hit a threshold of certainty.

This work motivated us to write an article, which has been accepted and published, for the LowCode 2021 workshop in the MODELS ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems, entitled "Improving Collaboration Efficiency Between UX/UI Designers and Developers in a Low-Code Platform" [80]. The submitted article can be seen in the Annex II.

Bibliography

[1] M. Abrams et al. "UIML: an appliance-independent XML user interface language". In: *Computer Networks* 31.11 (1999-05), pp. 1695–1708. ISSN: 13891286. DOI: 10.1016/S1389-1286(99)00044-4 (cit. on p. 24).

[2] Adobe. *Adobe Photoshop*. Photoshop. 2021. URL: https://www.photoshop.com (visited on 2021-02-19) (cit. on p. 13).

[3] *Adobe XD | Ferramenta de colaboração e design de UI/UX rápida e avançada*. Adobe. URL: https://www.adobe.com/br/products/xd.html (visited on 2021-02-19) (cit. on p. 14).

[4] Anima. *Anima | Design to development platform*. URL: https://www.animaapp.com/ (visited on 2021-02-10) (cit. on p. 21).

[5] C. Atkinson and T. Kuhne. "Model-driven development: a metamodeling foundation". In: *IEEE Software* 20.5 (2003-09), pp. 36–41. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231149 (cit. on p. 15).

[6] *Avocode App - Collaborate on Design Files with Anyone*. Avocode. URL: https://avocode.com/ (visited on 2021-02-19) (cit. on p. 14).

[7] S. B.V. *API Reference*. Sketch Developers. URL: https://developer.sketch.com/reference/api/ (visited on 2021-08-04) (cit. on p. 32).

[8] S. B.V. *Command-line interface*. Sketch Developers. URL: https://developer.sketch.com/cli/ (visited on 2021-08-04) (cit. on p. 32).

[9] S. B.V. *File format*. Sketch Developers. 2021. URL: https://developer.sketch.com/file-format/ (visited on 2021-02-09) (cit. on p. 34).

[10] S. B.V. *Layer Basics*. Sketch. URL: https://www.sketch.com/docs/layer-basics/ (visited on 2021-02-16) (cit. on p. 16).

[11] S. B.V. *Shapes*. Sketch. URL: https://www.sketch.com/docs/shapes/ (visited on 2021-02-18) (cit. on p. 16).

[12] S. B.V. *Symbols*. Sketch. URL: https://www.sketch.com/docs/symbols/ (visited on 2021-02-18) (cit. on p. 16).

[13] S. B.V. *The digital design toolkit*. Sketch. URL: https://www.sketch.com/ (visited on 2021-02-18) (cit. on pp. 11, 14, 16, 21, 22).

[14] *Balsamiq. Rapid, Effective and Fun Wireframing Software | Balsamiq*. URL: https://balsamiq.com/ (visited on 2021-02-19) (cit. on p. 14).

[15] T. Beltramelli. "pix2code: Generating Code from a Graphical User Interface Screenshot". In: *arXiv:1705.07962 [cs]* (2017-09-19). arXiv: 1705.07962. URL: http://arxiv.org/abs/1705.07962 (visited on 2021-01-18) (cit. on pp. 23, 25).

[16] M. Bexiga. "Closing the Gap Between Designers and Developers in a Low-Code Ecosystem". Master's Thesis. NOVA School of Science & Technology, 2021 (cit. on pp. 9, 10, 12, 18, 21, 23).

[17] M. Bexiga, S. Garbatov, and J. C. Seco. "Closing the gap between designers and developers in a low code ecosystem". In: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020-10-16, pp. 1–10. ISBN: 978-1-4503-8135-2. DOI: 10.1145/3417990.3420195 (cit. on pp. 12, 19, 22, 23, 25).

[18] F. Budinsky et al. "Automatic Code Generation from Design Patterns". In: *IBM Systems Journal* (1996), pp. 151–171 (cit. on p. 21).

[19] C. Candeias. *6 Steps to Improve User Experience: Embracing Your End-Users*. 2020-01-30. URL: https://www.outsystems.com/blog/posts/improve-user-experience/ (visited on 2021-07-02) (cit. on p. 1).

[20] J. M. Carroll. *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*. Google-Books-ID: gGyEOjkdpbYC. Elsevier, 2003-05-21. 579 pp. ISBN: 978-0-08-049141-7 (cit. on p. 24).

[21] U. Census. *Population Clock: World*. 2021. URL: https://www.census.gov/popclock/world (visited on 2021-02-08) (cit. on p. 1).

[22] D. Data. *New Research from Dimension Data Reveals Uncomfortable CX Truths*. 2017-04-04. URL: https://www.prnewswire.com/news-releases/new-research-from-dimension-data-reveals-uncomfortable-cx-truths-300433878.html (visited on 2021-02-08) (cit. on p. 1).

[23] A. van Deursen, P. Klint, and J. Visser. "Domain-specific languages: an annotated bibliography". In: *ACM SIGPLAN Notices* 35.6 (2000-06-01), pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035 (cit. on p. 15).

[24] M. Dixon and J. Fogarty. "Prefab: Implementing Advanced Behaviors Using Pixelbased Reverse Engineering of Interface Structure". In: *In Proceedings of the ACM Conference on Human Factors in Computing Systems*. 2010 (cit. on pp. 22, 25).

[25] S. Faraj and L. Sproull. "Coordinating Expertise in Software Development Teams". In: *Management Science* 46 (2000-12-01), pp. 1554–1568. DOI: 10.1287/mnsc.46.12.1554.12072 (cit. on p. 1).

[26] K. C. Feldt. *Programming Firefox: Building Rich Internet Applications with XUL*. 1st edition. O'Reilly Media, 2007-04-25. 513 pp. (cit. on p. 24).

[27] Figma. *Figma: the collaborative interface design tool*. Figma. URL: https://www.figma.com/ (visited on 2021-02-18) (cit. on pp. 14, 21, 22).

[28] B. Frost. *Atomic Design*. Google-Books-ID: 1e92vgAACAAJ. Brad Frost Web, 2016-12-05. 193 pp. ISBN: 978-0-9982966-0-9 (cit. on p. 6).

[29] B. Frost. *Atomic Design Methodology | Atomic Design by Brad Frost*. URL: https://atomicdesign.bradfrost.com/chapter-2/ (visited on 2021-02-18) (cit. on p. 8).

[30] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock. "Automatically generating personalized user interfaces with Supple". In: *Artificial Intelligence* 174.12 (2010-08), pp. 910–950. ISSN: 00043702. DOI: 10.1016/j.artint.2010.05.005 (cit. on p. 21).

[31] *GIMP*. GIMP. URL: https://www.gimp.org/ (visited on 2021-02-19) (cit. on p. 13).

[32] K. Greff et al. "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (2017-10). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 2222–2232. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2016.2582924 (cit. on p. 23).

[33] T. Grossman and R. Balakrishnan. "The bubble cursor: enhancing target acquisition by dynamic resizing of the cursor's activation area". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '05. New York, NY, USA: Association for Computing Machinery, 2005-04-02, pp. 281–290. ISBN: 978-1-58113-998-3. DOI: 10.1145/1054972.1055012 (cit. on p. 22).

[34] S. Hassan et al. "Extraction and Classification of User Interface Components from an Image". In: *International Journal of Pure and Applied Mathematics* 118 (), p. 16. ISSN: 1314-3395 (cit. on pp. 23, 25).

[35] J. Hutchinson et al. "Empirical assessment of MDE in industry". In: *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. Proceeding of the 33rd international conference. Waikiki, Honolulu, HI, USA: ACM Press, 2011, p. 471. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985858 (cit. on p. 15).

[36] F. Inc. *React – A JavaScript library for building user interfaces*. 2021. URL: https://reactjs.org/ (visited on 2021-09-02) (cit. on p. 5).

[37] I. Inc. *Design-to-code | Design Defined | InVision*. Design-to-code | Design Defined | InVision. URL: https://www.invisionapp.com/design-defined/design-to-code/ (visited on 2021-02-11) (cit. on pp. 2, 14).

[38] *InVision Inspect*. InVision. URL: https://www.invisionapp.com/feature/inspect (visited on 2021-02-19) (cit. on p. 14).

[39] V. Jain et al. "Sketch2Code: Transformation of Sketches to UI in Real-time Using Deep Neural Network". In: *arXiv:1910.08930 [cs, eess]* (2019-10-20). arXiv: 1910.08930. URL: http://arxiv.org/abs/1910.08930 (visited on 2021-01-18) (cit. on pp. 23, 25).

[40] J. Jelinek and P. Slavik. "GUI generation from annotated source code". In: *Proceedings of the 3rd annual conference on Task models and diagrams*. TAMODIA '04. New York, NY, USA: Association for Computing Machinery, 2004-11-15, pp. 129–136. ISBN: 978-1-59593-000-2. DOI: 10.1145/1045446.1045470 (cit. on p. 21).

[41] K. Jiang. *Introducing: Figma to React*. Figma. 2018-04-26. URL: https://www.figma.com/blog/introducing-figma-to-react/ (visited on 2021-02-16) (cit. on p. 22).

[42] J. Johnson. *Internet users in the world 2020*. Statista. 2021-01-27. URL: https://www.statista.com/statistics/617136/digital-population-worldwide/ (visited on 2021-02-08) (cit. on p. 1).

[43] S. Kelly and J.-p. Tolvanen. *1 Visual domain-specific modelling: Benefits and experiences of using metaCASE tools* (cit. on p. 15).

[44] S. W. J. Kozlowski and D. R. Ilgen. "Enhancing the effectiveness of work groups and teams". In: *Psychological Science Suppl. S* (2006), pp. 77–124 (cit. on p. 1).

[45] K. Kristensen and B. Kijl. "Collaborative Performance: Addressing the ROI of Collaboration". In: *International Journal of e-Collaboration* 6.1 (2010-01), pp. 53–69. ISSN: 1548-3673, 1548-3681. DOI: 10.4018/jec.2010091104 (cit. on p. 1).

[46] T. Kühne. "Matters of (Meta-) Modeling". In: *Software & Systems Modeling* 5.4 (2006-11-22), pp. 369–385. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-006-0017-9 (cit. on p. 15).

[47] Y. Lee, K. A. Kozar, and K. R. Larsen. "The Technology Acceptance Model: Past, Present, and Future". In: *Communications of the Association for Information Systems* 12 (2003). ISSN: 15293181. DOI: 10.17705/1CAIS.01250 (cit. on p. 55).

[48] M. B. Lieberman and D. B. Montgomery. "First-mover advantages". In: *Strategic Management Journal* 9 (S1 1988), pp. 41–58. ISSN: 1097-0266. DOI: https://doi.org/10.1002/smj.4250090706 (cit. on p. 3).

[49] Q. Limbourg et al. "USIXML: A Language Supporting Multi-path Development of User Interfaces". In: *Engineering Human Computer Interaction and Interactive Systems*. Ed. by R. Bastide, P. Palanque, and J. Roth. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 200–220. ISBN: 978-3-540-31961-0. DOI: 10.1007/11431879_12 (cit. on p. 24).

[50] Y. Lindsjørn et al. "Teamwork quality and project success in software development: A survey of agile development teams". In: *Journal of Systems and Software* 122 (2016-12-01), pp. 274–286. ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.09.028 (cit. on p. 1).

[51] d. LLC. *Paint.NET - Free Software for Digital Photo Editing*. URL: https://www.getpaint.net/index.html (visited on 2021-02-19) (cit. on p. 13).

[52] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[53] N. Ltd. *React Studio*. URL: https://reactstudio.com/ (visited on 2021-02-10) (cit. on p. 21).

[54] J. Ludewig. "Models in software engineering ? an introduction". In: *Software and Systems Modeling* 2.1 (2003-03-01), pp. 5–14. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-003-0020-3. (Visited on 2020-11-20) (cit. on p. 15).

[55] J. Luoma, S. Kelly, and J.-p. Tolvanen. "Defining Domain-Specific Modeling Languages: Collected Experiences". In: *In Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM04*. 2004 (cit. on p. 15).

[56] M. Macik. "Automatic User Interface Generation". PhD thesis. 2016-06-29. DOI: 10.13140/RG.2.2.23963.26401 (cit. on p. 23).

[57] M. Macık. "Automatic user interface generation". PhD thesis. PhD thesis, Faculty of Electrical Engineering, Czech Technical University, 2016 (cit. on p. 21).

[58] L. A. MacVittie. *XAML in a Nutshell*. Google-Books-ID: v03elGOy9ogC. "O'Reilly Media, Inc.", 2006. 302 pp. ISBN: 978-0-596-52673-3 (cit. on p. 24).

[59] T. Mens and P. Van Gorp. "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005) 152 (2006-03-27), pp. 125–142. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.10.021 (cit. on p. 15).

[60] Modulz. *Modulz*. URL: https://modulz-website.now.sh/ (visited on 2021-02-10) (cit. on p. 21).

[61] NASA. *TLX @ NASA Ames - Home*. URL: https://humansystems.arc.nasa.gov/groups/tlx/ (visited on 2021-07-05) (cit. on p. 55).

[62] Newtonsoft. *Json.NET - Newtonsoft*. URL: https://www.newtonsoft.com/json (visited on 2021-02-04) (cit. on p. 35).

[63] T. A. Nguyen and C. Csallner. "Reverse Engineering Mobile Application User Interfaces with REMAUI (T)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2015-11, pp. 248–259. DOI: 10.1109/ASE.2015.32 (cit. on pp. 22, 25).

[64] J. Nielsen. *Paper Prototyping: Getting User Data Before You Code*. Nielsen Norman Group. URL: https://www.nngroup.com/articles/paper-prototyping/ (visited on 2021-02-16) (cit. on p. 23).

[65] D. Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Revised edition. New York, New York: Basic Books, 2013-11-05. 368 pp. ISBN: 978-0-465-05065-9 (cit. on p. 2).

[66] D. A. Norman. *The Design of Everyday Things*. Google-Books-ID: b09jQgAACAAJ. Doubleday, 1990. 257 pp. ISBN: 978-0-385-26774-8 (cit. on p. 1).

[67] S. O'Dea. *Smartphone users 2020*. Statista. 2020-12-10. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (visited on 2021-02-08) (cit. on p. 1).

[68] *Optimize design files for developer handoff*. Figma. URL: https://help.figma.com/hc/en-us/articles/360040521453-Optimize-design-files-for-developer-handoff (visited on 2021-02-19) (cit. on p. 14).

[69] Oracle. *Seventy-Seven Percent Of Consumers Feel Inefficient Customer Service Experiences Detract From Their Quality of Life*. 2018-04-10. URL: https://www.prnewswire.com/news-releases/seventy-seven-percent-of-consumers-feel-inefficient-customer-service-experiences-detract-from-their-quality-of-life-300626778.html (visited on 2021-02-08) (cit. on p. 1).

[70] OutSystems. *Banco Santander Consumer Portugal Adopts OutSystems as Their Digital Transformation Platform*. URL: https://www.outsystems.com/case-studies/santander-consumer-digital-transformation/ (visited on 2021-02-19) (cit. on p. 5).

[71] OutSystems. *Extensions*. OutSystems. 2018-07-26. URL: https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/Extend_Logic_with_Your_Own_Code/Extensions (visited on 2021-02-18) (cit. on p. 20).

[72] OutSystems. *Hospital Management System Deploys in Weeks, Not Years*. URL: https://www.outsystems.com/case-studies/hospital-management-system/ (visited on 2021-02-19) (cit. on p. 5).

[73] OutSystems. *Libraries*. OutSystems. 2019-07-26. URL: https://success.outsystems.com/Documentation/11/Developing_an_Application/Reuse_and_Refactor/Libraries (visited on 2021-02-18) (cit. on p. 20).

[74] OutSystems. *Live Style Guide Homepage*. URL: https://outsystemsui.outsystems.com/WebStyleGuidePreview/Homepage.aspx (visited on 2021-02-18) (cit. on p. 6).

[75] OutSystems. *Logitech Legacy Modernization Speeds Products to Market With OutSystems*. URL: https://www.outsystems.com/case-studies/fast-development/ (visited on 2021-02-19) (cit. on p. 5).

[76] OutSystems. *Mazda to Save Over $57M on Legacy Migration with OutSystems*. URL: https://www.outsystems.com/case-studies/mazda-legacy-migration-to-save-millions/ (visited on 2021-02-19) (cit. on p. 5).

[77] OutSystems. *OutSystems UI Framework: Past, Present, and Future*. URL: https://www.outsystems.com/blog/posts/ui-framework-future/ (visited on 2021-02-16) (cit. on pp. 5, 7).

[78] OutSystems. *Service Studio Overview*. OutSystems. 2019-03-07. URL: https://success.outsystems.com/Documentation/11/Getting_started/Service_Studio_Overview (visited on 2021-02-18) (cit. on pp. 19, 20).

[79] OutSystems. *UIPatterns | Live Style Guide*. 2021. URL: https://outsystemsui.outsystems.com/WebStyleGuidePreview/UIPatterns.aspx (visited on 2021-02-16) (cit. on p. 6).

[80] J. Pacheco, S. Garbatov, and M. Goulão. "Improving Collaboration Efficiency Between UX/UI Designers and Developers in a Low-Code Platform". In: *Proceedings of the 24rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '21, TBD. DOI: 978-1-6654-2484-4 (cit. on p. 66).

[81] Pagedraw. *Pagedraw/pagedraw*. original-date: 2019-01-12T20:13:39Z. 2021-02-11. URL: https://github.com/Pagedraw/pagedraw (visited on 2021-02-11) (cit. on p. 21).

[82] T. Palmer. *2019 Design Tools Survey Results*. URL: https://uxtools.co/survey-2019/ (visited on 2021-02-16) (cit. on p. 22).

[83] T. Palmer. *2020 Tools Survey Results*. 2020-12-01. URL: https://uxtools.co/survey-2020/ (visited on 2021-02-10) (cit. on pp. 14, 22, 23).

[84] F. Paterno', C. Santoro, and L. D. Spano. "MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments". In: *ACM Transactions on Computer-Human Interaction* 16.4 (2009-11-27), 19:1–19:30. ISSN: 1073-0516. DOI: 10.1145/1614390.1614394 (cit. on p. 24).

[85] C. Pemberton. *Key Findings From the Customer Experience Survey*. Gartner. 2018-03-16. URL: https://www.gartner.com/en/marketing/insights/articles/key-findings-from-the-gartner-customer-experience-survey (visited on 2021-07-02) (cit. on p. 1).

[86] *Pidoco - The Rapid Prototyping Tool*. Pidoco. URL: https://pidoco.com/en (visited on 2021-02-19) (cit. on p. 14).

[87] PixelCut. *PaintCode - Turn your drawings into Objective-C or Swift drawing code*. 2021. URL: https://www.paintcodeapp.com/ (visited on 2021-02-10) (cit. on p. 21).

[88] PixelCut. *PaintCode Plugin for Sketch*. 2021. URL: https://www.paintcodeapp.com/sketch (visited on 2021-02-16) (cit. on p. 22).

[89] A. Puerta and J. Eisenstein. "XIML: A Universal Language for User Interfaces". In: (2002-02-09) (cit. on p. 24).

[90] T. Puthiyamadam and J. Reyes. *Experience is everything: Here's how to get it right*. PwC. 2017-02-08. URL: https://www.pwc.com/us/en/zz-test/assets/pwc-consumer-intelligence-series-customer-experience.pdf (visited on 2021-02-08) (cit. on p. 1).

[91] *Reuse and Refactor*. OutSystems. 2018-07-26. URL: https://success.outsystems.com/Documentation/11/Developing_an_Application/Reuse_and_Refactor (visited on 2021-02-09) (cit. on p. 19).

[92] C. Richardson. *New Development Platforms Emerge For Customer-Facing Applications*. 2014-06-09. URL: https://www.forrester.com/report/New+Development+Platforms+Emerge+For+CustomerFacing+Applications/-/E-RES113411 (visited on 2021-02-18) (cit. on pp. 3, 21).

[93] A. Rodrigues da Silva. "Model-driven engineering: A survey supported by the unified conceptual model". In: *Computer Languages, Systems & Structures* 43 (2015-10-01), pp. 139–155. ISSN: 1477-8424. DOI: 10.1016/j.cl.2015.06.001 (cit. on p. 15).

[94] StatCounter. *Desktop Operating System Market Share Worldwide*. StatCounter Global Stats. URL: https://gs.statcounter.com/os-market-share/desktop/worldwide (visited on 2021-08-04) (cit. on p. 33).

[95] S. Studio. *Supernova*. Supernova. URL: https://supernova.io/ (visited on 2021-02-10) (cit. on p. 21).

[96] . *Svelte • Cybernetically enhanced web apps*. URL: https://svelte.dev/ (visited on 2021-09-02) (cit. on p. 5).

[97] Sympli. *Design Collaboration, Version Control & Handoff*. Sympli. URL: https://sympli.io (visited on 2021-02-19) (cit. on p. 14).

[98] M. Woo. "The Rise of No/Low Code Software Development—No Experience Needed?" In: *Engineering (Beijing, China)* 6.9 (2020-09), pp. 960–961. ISSN: 2095-8099. DOI: 10.1016/j.eng.2020.07.007 (cit. on pp. 3, 21).

[99] Yotako. *Seamless transition from design to code*. URL: https://www.yotako.io/ (visited on 2021-02-10) (cit. on p. 21).

[100] E. You. *Vue.js*. 2021. URL: https://vuejs.org/ (visited on 2021-09-02) (cit. on p. 5).

[101] Zeplin. *Zeplin*. Zeplin. URL: https://zeplin.io/ (visited on 2021-02-19) (cit. on p. 14).

I

Interview Questions

I.1 Designer Questions

- Do you receive any mockups from the client?
  - Yes:
    - What do you do with them? Do you start from scratch or do you modify them?
    - Do they also come in Sketch?
- The UI elements you create, do you do them from a reference point?
  - Yes:
    - Do you copy them from a repository or template?
    - Is there any "main model"?
    - How often is it updated?
  - No:
    - Would you be against the creation of one?
    - How disruptive would it be?
- What do you deliver to the front-end team?
- How many Sample Pages do you design?
- Can you walk me through the process of creating Sample Pages, as detailed as you can?
- What is the frequency of "custom"elements you create for a project?
- Is there any restriction you keep in mind during your work due to the design part?
- Suppose you wanted to create a custom component, how disruptive would it be if you had to add a field to the element marking it as custom?
- What is the thing that you'd find most disruptive/what you don't want to happen?

I.2 Developer Questions

- What do you receive from the design team?
- How many people are assigned per project?
  - If more than one:
    - How do you separate the roles?
- What/How is your workflow?
- What are your biggest challenges?
- Can you describe to me, to the smallest detail, the process of creating Sample Pages?
- The fact that you are restricted to 2 or 4 pages per project is it due to the complexity of the pages in account to the time you have?
- What are your criteria to choose the Sample Pages to make?
- Do you manually make everything just by looking at the designs?
  - Yes:
    - Are there any patterns/practices that help you identify the components in the pages?
- In this tool that I'm investigating, what would you say is essential or that you hope it accomplishes. What are the concerns you have and what is the worst thing that could happen?

II

MODELS 2021 Article

Improving Collaboration Efficiency Between UX/UI Designers and Developers in a Low-Code Platform

João Pacheco
NOVA School of Science and Technology
Portugal
jr.pacheco@campus.fct.unl.pt

Stoyan Garbatov
OutSystems
Portugal
stoyan.garbatov@outsystems.com

Miguel Goulão
NOVA School of Science and Technology,
NOVA LINCS
Portugal
mgoul@fct.unl.pt

***Abstract*—Customer-facing applications are essential for businesses. Therefore, a good user experience is fundamental for their success in the market. Companies employ highly specialized people in front-end development and User Experience (UX) & User Interface (UI) design to achieve this goal. Their collaboration is critical and raises some efficiency challenges, particularly in Low-Code platforms, such as OutSystems. UX/UI designers typically use specialized tools with their underlying metamodels. OutSystems developers use an integrated development environment with the underlying OutSystems metamodel. While there are some code-generation plugins for popular design tools, these do not generate code for Low-Code platforms.**

The current transformation process from design to development is done 100% manually, resulting in a loss of efficiency. Our goal is to accelerate this transformation process from a design model to a development model to mitigate this inefficiency.

To do so, we developed an approach using model transformations to automate part of the process. Namely, it automates the generation of application pages/screens by composing the screen mockups in a design technology (such as Figma or Sketch) with a library of reusable UI components to instantiate the design in a front-end technology (such as OutSystems).

Our approach was validated by a professional team of front-end developers from an established enterprise-grade Low-Code platform who applied and evaluated this work on some of their past real projects. Preliminary results show an overall acceptance of the developed tool with a possible increase of 150% to 400% in the number of pages/screens that they can deliver with the same effort.

This approach allows mitigating a bottleneck faced by the development team. To increase the value they could offer to customers (e.g., by producing more application screens in the same time frame), they would need to recruit new collaborators whose skill set is high on demand. Our work offers a more economical alternative to increase their productivity.

***Index Terms*—Design To Code, OutSystems, Low-Code Platforms, Front-end Development, Automation, Generation**

I. Introduction

The number of people using digital devices and applications has increased considerably [1], [2]. Smartphone usage has tripled from 2012 to 2019 to 3.2 billion with a prediction of 3.8 billion users by 2021 [3]. End-users expectations when interacting with a digital application, such as its usability and performance are also continuously growing [4]. Customer Experience (CX) is a significant factor in the market and can set a product apart from the competition [5]–[7]. The concept of Customer Experience includes many different aspects, one of them being Design. Gartner has reported that more than two-thirds of companies now "compete primarily on the basis of customer experience" [8].

Collaboration between people of diverse expertise and professional qualifications has been a general necessity for every industry [9], [10] and the software development industry is no exception. A research by Faraj and Sproull [11] has shown that the importance of coordinating well different expertises can overtake, in relevance, the existence of these expertises in the first place. A product's quality and value often reflect the teamwork between a combination of different roles. In 2016, Lindsjørn et al, investigated the relation between teamwork quality and the quality of the product, focusing more on teams that employed an agile development strategy, through a survey. This research concluded that the quality of teamwork plays a significant role, not only in terms of optimizing the team's performance, but also in increasing the quality of the product [12].

Developing software requires work and input by collaborators with distinct profiles and skills. A particularly relevant collaboration is that between the UX/UI design and front-end development practices. Designers design the entire system's expected behavior, look, and feel while developers turn the designs into reality through a front-end technology. One role cannot create the entire system at an enterprise-grade level. This means the collaboration between these areas is key to developing a successful product. UX/UI designers use specialized tools such as Sketch [13], or Figma [14] to create High- and Low-Fidelity prototypes of the products they are designing. Such tools are feature-rich and produce prototypes which can be seen as instantiations of their underlying metamodel. Some of these tools even offer code generation functionalities through plugins such as Anima [15]. Front-end developers take as input the prototypes from UX/UI designers and implement those prototypes with an Integrated Development Environment (IDE) which is usually specialized to their technology and needs. In the case of a Low-Code platform such as OutSystems, front-end developers have to manually translate the prototypes from the design format to the front-end technology by looking at each component or inspecting them with "Engineering hand-off tools" [16]. These methods are very time-consuming, no matter the developer's

expertise.

This work focuses on the OutSystems ecosystem, a Low-Code platform that shares these inefficiencies in the collaboration between UX & UI designers and front-end developers. An applied use case will be targeted at the Customer Success Department of OutSystems, where front-end developers compose these pages manually. These developers manually compose the UI elements into the pages using their IDE while inspecting the designs' properties to make both correspond as best as possible. This manual process is very inefficient and error-prone, no matter the expertise of the developer.

Our approach uses an horizontal exogenous unidirectional model transformation [17] from a source model, in Sketch, to a target model, in OutSystems. We use an independent intermediate model as a bridge from the source to the target model, splitting our transformation into 2: Sketch to Intermediate Model and then Intermediate to OutSystems. This makes it so in the future, we can support other technologies such as Figma.

II. Related Work

The research on the automation of application creation from design inputs is relatively novel compared to the other methods of automatic generation from model based code generation [18]–[21]. However, these past few years have shown some advances in this field.

A. Current & emerging solutions in the market

There are several continuing efforts to minimize the gap between design and front-end development by converting the design into code. Some existing applications, such as Anima [15], Supernova [22], and Yotako [23], allow users to provide their designs from multiple design tools such as Sketch [13] or Figma [14]. Most of these tools only export to a selection of code intensive frameworks such as React or HTML/CSS. Other applications like React Studio [24], PaintCode [25], and PageDraw [26] are even more strict, as they require the design to be made in their built-in editor. These restrictions would prevent designers from using their dedicated design application of choice and clash with today's industry practices in the design field [27]. None of them supports the generation of Low-Code artifacts. Even upcoming tools such as Modulz [28] with state-of-the-art techniques to automatically generate code in the background, as the designers draw the designs, share this limitation.

B. Plugins

Sketch and Figma are among the most commonly used vector design tools globally [29], [30]. None of them has native support for exporting the designs into code. However, both can be extended via plugins [31], [32].

Unfortunately, neither of these functionalities are suitable for OutSystems since these code generation functionalities do not export to their Low-Code technology [33].

C. Methods to identify UI elements

There are several methods for supporting design-to-code automatic generation. They often use computer vision [34]–[37], or optical character recognition [38] to identify the patterns in the designs. Some of these methods also use neural networks and require much training to increase their viability. These approaches use images and hand-drawn sketches as input, making them somewhat disconnected from current established practices designers follow. While people still use pen and paper to hand draw ideas during brainstorming sessions [30] or initial testing using paper prototyping [39], it does not go much further. They often employed these strategies to save money and resources during the early stages of the development process. However, these strategies are time-consuming and leave out essential details of the system such as looks, feedback, overall feel, and other Human-Computer Interaction (HCI) essential elements [40]. Designers turn to these specialized design tools to construct a higher fidelity design incorporating these elements to get the system's feeling, look, and interactions. Bexiga *et al.* developed a method and a tool to automate the reusable UI components' styling process by transforming these components from an artifact created in a design tool into a Low-Code technology [27], [33]. This mitigated another inefficiency in the collaboration between designers and developers, making it the most related to our work. Their work was also applied to the OutSystems ecosystem. The tool achieved a time-save ranging from 2-3 days out of the 5 days front-end developers have to do their work. Our work, focuses on the composition of the reusable components to create application pages, while theirs applies to the styling of these components. We are also expecting different kinds of results. Bexiga's work is focused on shortening the time front-end developers spend on instantiating the reusable components while our goal is to support an increased efficiency of the front-end teams in developing Sample Pages, allowing them to deliver more sample pages with the same effort.

D. Representation of User Interfaces

User interfaces can be specified with User Interface Description Languages (UIDLs), which are usually XML-based. User Interface Markup Language (UIML) [41] is a platform-independent language for describing UI. Due to its high level of abstraction, other UIDLs, such as those discussed here, are often referred to as UIMLs instead of UIDLs.

Extensible Application Markup Language (XAML) [42] is an XML-based declarative language created by Microsoft for their .NET applications. Since this language is the basis of Microsoft's .NET applications, it contains the User Interface structure and embeds programming logic and styling. Every XAML tag corresponds to a .NET component whose properties can be controlled through the tags' properties.

The Mozilla Foundation created XML User Interface Language (XUL) [43]. It is structured similarly to Web Pages. Like XAML, XUL can be extended with existing standards and technologies such as CSS and JavaScript.

These presented UIDLs (and others) were all created with the goal of representing UI in XML based representations and are specialized in frameworks. We will be needing a similar representation to these that fits more in the lines of the artifacts created by Design Tools and Low-Code front-end technologies. We could try to pick one of these and try to adapt them into our needs but ultimately we decided to create our own. Our representation will differ from most UIDLs as it will be JSON-based instead of XML since we will be working with JSON throughout the process and have everything already setup.

III. Approach

A. Initial Concerns

Before implementing the approach, we set out a couple of guidelines we believe are essential for such an approach. These were as follows:

- The tool should be as unintrusive as possible, to facilitate its adoption among UX/UI designers, so as not to disrupt their workflow, while providing a significant head-start for front-end developers, freeing their time.
  - If any adjustments are needed, these should be minimal and not intrusive.
- The tool should be simple to use.

We also opted to not utilize any model transformation language and instead we will be working with their internal representations which come in the form of JSON files. This means our transformations will be based on rules that are hard-coded into the tool. We made this decision in order to be more pragmatic as this work is being made for a thesis with a time restraint.

Another concern is how aggressive we should be with the automation process, so we considered two alternatives:

A first approach would be to focus on **Precision** by only instantiating the components, and parts of them, when we are close to 100 % sure we are doing it correctly. The consequence of this is we end up with fewer components and more incomplete final results, but with greatly improved accuracy.

A second approach would be to focus on **Recall** as we automate as much as possible, even for ambiguous scenarios when we are not 100% sure of the appropriate process. This leads to a higher risk of incorrectly instantiated components, forcing the development team to reorganize or even redo from scratch.

We opted to focus on **Precision** with approach 1 since it causes less overhead on the development team and anything we do not create, they can easily do with their expertise.

B. High Level Overview

Our approach will be divided into 2 phases. The first phase is to transform the given Sketch file, namely the screen mockups it possesses, into an Intermediate Representation. The second phase is to pick these Intermediate Representation screens, and transform them into OutSystems screens, by composing them with a library of reusable UI components already previously instantiated in OutSystems called Live Style Guide. This will result in an OutSystems application with structured pages also known as Sample Pages. The pages are fully functional screens created by the Customer Success team for the customer. For each particular customer's business and business model, these pages are created and personalized. These pages are structured mainly by the reusable UI elements in the LSG and then are populated with mock data to allow the customer to test the system's interactions while serving as a stepping stone for the customer to build on. A representation of the process can be seen in Figure 1.

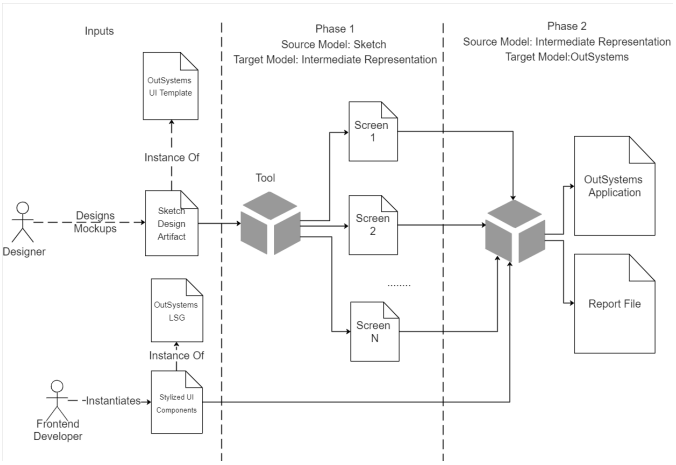


Fig. 1. High level overview of our solution

OutSystems employs these Live Style Guides, containing reusable UI components (such as buttons and menus) and Sample Pages for a specific application. Their goal is to foster consistency, improve the developers' overall efficiency and productivity when constructing screens for their application while lowering the required skill set to do so, and reduce the page's components' maintenance cost. They accomplish this by implementing a variation of the Atomic Design methodology proposed by Frost [44].

To better understand the ins and outs of these teams' workflows, we interviewed five members of the *Customer Success* department, namely two front-end experts and three designers, to complement what information we had from Bexiga's work [27]. This department is responsible for creating and delivering projects to OutSystems' clients. In addition, they provide various services, or "Packs". In this paper, we are particularly interested in the "Accelerator Packs." These "Packs" help new clients get started on their first application as the department provides them with a LSG. The flow in these services is usually divided into three main phases: *Kick-Off meeting*, *Design*, *Development*.

We are interested in the transition between the *Design* and the *Development* phases. At the end of the Design phase, all of the design for the application screens and components is often finished, and it is passed to the developer to instantiate them in the OutSystems language manually. Unfortunately, there are no

tools or mechanisms that help the front-end developer convert these designs. These designs are always made in Sketch, and the translation process is usually done in 5 days.

Another essential piece of information gathered from these interviews is that two different design artifacts are created in the Design phase as this phase can be divided into two sub-phases. In the first sub-phase, also known as the UX phase, the designer creates the UX design of tentatively every screen in the application (Low-Fidelity mockups) by composing the screens with components from a template called OutSystems UI LSG Template, and it usually takes ten days.

After these ten days end, the second sub-phase, the UI design phase, begins. The UI designer takes the Low-Fidelity mockups and transforms them into High-Fidelity mockups. In this phase, it is an established practice first to create the screens and then the LSG. This leads to the possibility of the components in the screens being decoupled, which makes it harder to identify these components as their direct connection to the source component in the template is lost.

Considering these insights, there are two possible solutions of which type of design artifacts to use: High-Fidelity or Low-Fidelity. Conceptually, we set out to focus on using the Low-Fidelity mockups as the input design artifact of choice as it is more "safe" from decoupled elements since its an established practice to use the components in the template, but ultimately, we support the use of both.

Finally, in the Development phase, the assigned Developer takes these High-Fidelity mockups and instantiates them. This is a 100% manual process that takes the course in 5 days - 3 dedicated to the library's construction and 2 for creating Sample Pages. The Sample Page creation process can be divided into three big steps:

1) Structure: Developers create the structure of the page by dragging and dropping the components onto the page.
2) Instantiation: Developers instantiate the components by assigning any necessary parameters, such as variables and auxiliary structures.
3) Data: Developers create logic that bootstraps sample domain data to populate the page and match the customer's business model.

We took step 1 as our primary goal for our work while checking the possibility of doing step 2, even if just partially.

C. Models in our Work

Before we discuss the implementation and the more in-depth details of the model transformations in our work, we will first discuss our source and target models.

1) Sketch Model: Sketch is a digital design toolkit [13]. Its primary focus is on producing Low and High-Fidelity representations for user interfaces. Internally, Sketch is composed of "Layers." These Layers are the building blocks for creating designs in Sketch [45]. The main Layer types are:

- **Group:** These are composed of multiple layers. There are specialized kinds of this layer type such as:
  - **Pages:** These are groups that represent a canvas of the document. A simplified way of visualizing these pages would be referring to them as categories, i.e., a Page called UI would contain every UI design.
  - **Artboard:** These contain the designs of what would be the actual screens of the system.
  - **Symbol Master:** These are group layers that are frequently used around the whole document. As mentioned above, any change applied to them is propagated in their instances [46].
  - **Symbol Instance:** These are instances of the Symbol Master and inherit their properties. However, these can alter properties independent of the master [46].
- **Text:** Layer type that contains textual information.
- **Shape:** The most common type of Layers in Sketch. They represent pre-made shapes that are pre-built or created by the designer[47].

Layers have properties that characterize their elements. Important ones to mention are:

- **Name:** Represents the name given by the designer.
- **Class:** Represents the layer (mentioned above).
- **SymbolID:** If the Class attribute is a SymbolInstance, then it will have a SymbolID which corresponds to its Symbol Master
- **Frame:** Defines the positions and dimensions of the layer relative to the parent layer.
- **Layers:** List composed of sub-layers.

2) OutSystems UI Template Sketch File Model: OutSystems' UX designers use a template called *OutSystems UI* whenever doing Low-Fidelity mockups. This facilitates the mapping process later on, as it would not be possible to do so without guaranteeing the design mockups' structure. This file was developed by OutSystems' UI team [33] and is continuously updated to match the evolution of the OutSystems UI framework. During our research, an issue around the usage of this file arose, due to its continually evolving state: some designers are using different versions of this file. We will require the design team to utilize the same updated template file for our work to function, as our prototype is hardwired to the current version (at the time of writing this article) of the template. Not doing so could show some inaccuracies or odd behaviors.

The template's structure is divided into the following Pages:

- **Style:** Represents the common styling properties such as typography, color, and shadow. This page is ignored in our automatic process as we do not want to hard-wire this styling information into the generated components. Instead, this page will be used by the Developer in the future to create the necessary CSS classes to allow for more modularity and easier maintenance.
- **UI Patterns and Widgets:** This page contains every UI pattern and widget and their respective state (on, off, primary, secondary, loading, and so on) and device (web, mobile, tablet) variations in the OutSystems UI kit. This is another page that we do not use as there is no link from the designs to this page. The purpose of it, is to

help the Developer create the variations of the widgets in OutSystems.

- **Layouts:** The page shows the different layouts of the pages for the different possible devices, such as a desktop, tablet, mobile, and screen sizes. This will be used by the developer to create the respective "main layout" which we will use to place our components.
- **Symbols:** It contains every Symbol Master that's going to be instantiated in the designs. This page is important for us as we will use it as our ground-truth to correctly identify the components.

3) Intermediate Representation Model: We created our own model to bridge our source and target models to achieve a higher abstraction level. Our model is mostly based on the Sketch model, with additional information to support a correct mapping to OutSystems.

The base element is an IRNode, an abstract structure created by us to serve as the nodes of our Intermediate Representation, which follows a tree-like structure. There will be an IRNode per supported Sketch layer containing all the information that is common in every component, such as:

- **Type:** Represents the type of the node. It can be a group, symbol, or root, determined by the *Class* of the layer.
- **InternalName:** A string that represents the name of the component in the Live Style Guide.
- **Widget:** A simplified version of InternalName. It allows to easily distinguish components, i.e., an InternalName like `01.adaptable/component/Button Default/Primary` goes to Primary Button.
- **DesignerName:** The layer name, as chosen by the designer.
- **RelativePosition:** Contains the coordinates of the component in relation to the parent component, its width and height.
- **Children:** A list containing the descendant layers. Useful for nested components.
- **Visited:** An auxiliary Boolean to help against duplicates.

Besides these common attributes, every component has specific attributes, such as the number of columns in a Columns widget or the font size in a Text component. Fig 2 shows a fragment of this model including three different components.

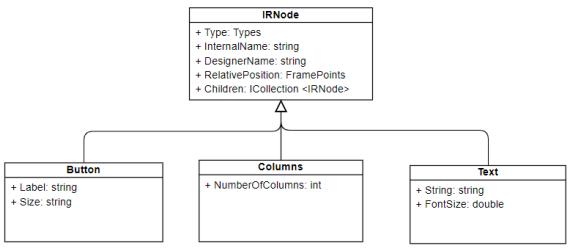


Fig. 2. Fragment of the intermediate representation model

4) OutSystems Model: The OutSystems platform employs a Visual Programming Language that abstracts an underneath strongly typed language [48] that composes its four significant layers: processes, interfaces, logic, and data [49]. In our work, we will focus on the interface layer since we are focusing on composing the UI's structure and, as the name suggests, this layer is related to the UI of an OutSystems project. OutSystems' applications are composed of various modules to help reusability and maintainability. Each module is separated in its Service Studio Document (OML) file. These files are proprietary to OutSystems and contain the information of modules. To see this information, you need a specialized tool created by OutSystems that extracts the information in these OML files into Extensible Markup Language (XML) format. There are four kinds of modules: application, service [49], library [50], and extension [51]. The application's screens are called **Web Screens** and are composed of various UI elements representing a hierarchical structure. These UI elements can be separated into multiple categories, such as "widgets," whose primary goal is for the user to interact with the system by inputting or submitting. An example of these UI components are buttons and dropdown menus. OutSystems also has WebBlocks which are reusable groups of various UI components often created due to it being common in various screens. With these WebBlocks, users can take advantage of the benefits in Atomic Design.

Table I maps Sketch elements to the OutSystems model.

	Sketch	OutSystems
Internal Structure	Hierarchical Structure of Layers	Hierarchical Structure of Widgets
Application Screens	Artboard	Web Screen
Atomic Elements	Symbols	LSG Elements (UI Patterns)
Element Grouping	Groups	Containers or Web Blocks

TABLE I
MAPPINGS BETWEEN SKETCH AND OUTSYSTEMS

D. Sketch to Intermediate Representation

The first challenge was identifying the components in a Sketch artboard. The design practice is inherently heterogeneous, meaning different designers build their designs differently. The same designer can do things differently in different designs, or even in the same one. Our tool's effectiveness is heavily dependent on how these were instantiated, constraining our ability to identify the components accurately.

With a few exceptions (e.g. links and forms), the OutSystems UI Sketch template has the components in the OutSystems library set up as Symbols. Designers can select the Symbol and paste it into a design, creating a Symbol Instance. These Symbol Instance layers have an ID that allows tracing them to the Symbol Master, letting us know what this layer is referencing regardless of whatever modifications a designer does. However, when modifying these instances, designers have some restrictions. For example, changing text in a Symbol is allowed, but nesting components, e.g., adding a Button in a Card, is not allowed. Designers can either create this nested component from scratch or decouple the Symbol Instance so it turns into a group layer (we would do this to

the Card), preserving everything the component has except the reference to the Master Symbol. Then we can nest the other component. In our implementation, when we discover a group, we always check if it is a decoupled component by its name, which is preserved when decoupling components.

1) Preparation Phase: To access and manipulate a Sketch file, we need to unzip the file [52]. This results in various folders (Fig. 3) , where one of them contains every artboard in the Sketch document in a JSON format. Firstly we need to locate the JSON that corresponds to the designs. For this to work, the pages' name needs to be constant, defined and used across any project that will use this tool, so we can precisely identify the page to be used. We are looking for the ones that correspond to the UX Design (Low-Fidelity mockups) and the UI Design (high-fidelity mockups). If only one exists, then we will use that page. If both exist, we have a priority method to prioritize the UX Design for reliability concerns, as it is an established practice to use the template's components to model this kind of design which allows us to identify the components with high precision.

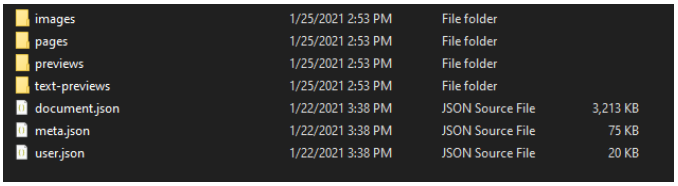


Fig. 3. Example of the structure of a Sketch file after unzipping

2) Layer Treatment & Identification: Once found, we go through every sub-layer that is of class `artboard` as these will be our application screens. Each of these `artboard` layers contains many attributes that are not relevant for our model transformation. For each layer and respective sublayers, we do a "cleanup" by removing everything except for the set of attributes we need.

With these "simplified" artboards, we do a depth-first traversal of their layers and sublayers. Thanks to the class attribute on these layers, we can check what each layer represents. If it is a `symbolInstance`, we take its `symbolID` field and use an auxiliary structure to identify its corresponding element. Things get more complicated if it is a `group` layer. With the help of an auxiliary structure we can see if it is a decoupled component or a simple group by checking the layer's name. If the designer changes the name of a decoupled component, we can no longer guarantee a connection between this layer and its reference.

3) Intermediate Representation Node: Regardless of its class (`group` or `symbolInstance`), the layer will be handled and used to create an `IRNode`, the base element of our Intermediate Representation. This representation follows a tree-like structure composed of IRNodes. Every element, be it a Symbol layer that will map to an OutSystems element or a text layer, will have its node class extending the abstract base class as each component will require different treatment. For example, a Button will have its class with the required properties and processes to be implemented correctly in OutSystems. This makes it easier to add support for more components.

Fig. 4. Text representation in Sketch.

Fig. 5. Text representation in Intermediate Representation.

We currently support **20** components of the OutSystems LSG. These were identified by Front-End experts at OutSystems as the most relevant components.

Observing Figures 4 & 5, we can compare the structure of an artboard in the Sketch model and our Intermediate Representation. Some significant takeaways are:

1) The code snippets show a simple conversion of a Button component.
2) Different components have different properties. For example, label in buttons, as seen in figure 5 does not appear in the other component nodes.
3) Only necessary attributes for structuring purposes were kept.

Our second most significant challenge was handling horizontal placement and representing it in our Representation. The OutSystems language provides users two different ways to place UI components horizontally. The first one is using a simple container with the `display: flex` CSS property. The second way is to use the Columns widget. The Columns widget creates x number of placeholders for the user to put the content they wish to display and automatically handles the spacing between columns at the cost of some performance. We developed an algorithm that identifies which elements overlap horizontally and groups them in either a Columns node or a group flex node, depending on the group's components.

The output of this process is a JSON file per application screen in our Intermediate Representation model.

E. Intermediate Representation to OutSystems

We use OutSystems' ModelAPI to create and modify OutSystems applications programmatically to create the application's screens. With this API, we will reference the elements already customized and instantiated by the Front-End developer that come in an OML file. OutSystems files have keys that are generated and are essential in the inner structure of these OML files. These keys have already been created and reside in the given OML file, so we only edit the source file by injecting the generated Screens into it, preserving these keys.

We create a Screen Flow (how OutSystems groups application screens) called `GeneratedSamplePages`, where we will store every Web Screen that we create for each JSON file created. After creating a Web Screen, our tool does a depth-first analysis of our Intermediate Representation. After serializing these components from the JSON, we generate a Container widget for each node to encapsulate them and then instantiate the node. Each node has its specific way of instantiating, but they fall in 3 major groups:

1) ModelAPI provides direct support for basic widgets (e.g. Buttons, Dropdowns), making instantiation trivial.
2) More advanced widgets such as the Card or Columns, require fetching the widget through references.
3) Group widgets can have two variations that vastly change how they work:
   - We instantiate a container for simple groups and instantiate subnodes together in said Container.
   - Flex groups require horizontal placement, so we create a Container encapsulating all the components in the group with the property `display: flex`; each component then has a container for itself.

In the end, an OML file will be output with the respective application with the screens fully structured and preserving sensitive internal data such as the keys mentioned above, and a report file will mark any issues found and helpful information for the developers to solve such problems.

The main challenges in this part of the project derived from the early stage in which the ModelAPI is. As it is still in development, there is no documentation and accessing some properties is still not as straightforward as one would wish. We implemented a way to have inline styling with fixed widths and heights to look closer to the original designs. Still, ultimately we had to take them out as it would not allow for a responsive behavior as is their goal and would give them more work.

Figure 6 showcases a more in-depth and complete overview of our approach.

F. Sub-Goal Validation

When implementing support for the first set of the 20 components, we also checked the possibility of doing the secondary goal: instantiating part of the components, such as assigning them necessary auxiliary variables. We decided to drop this goal and focus solely on the primary as this one presented a bad return on investment. Although the ModelAPI allows instantiating the necessary variables and structures and assigning them, we cannot detect any interaction between components from the designs, so the concept of shared variables disappears. For example, clicking a button to add 1 to the number present on a text box. These components share a variable that is being modified by the button and presented by the text field.

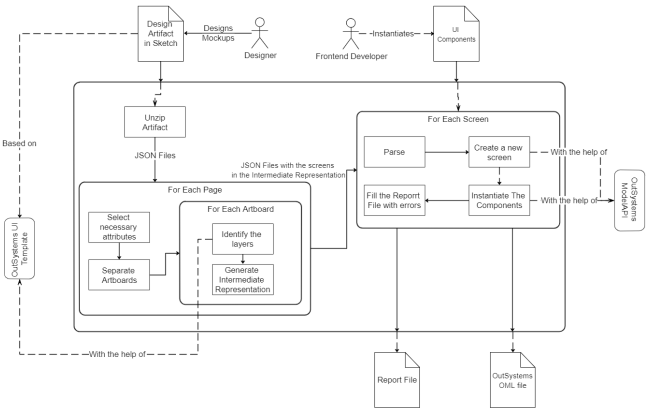


Fig. 6. Diagram of the developed approach.

This would lead us to only one possible solution: creating a new variable per component that requires it. Doing it this way would most certainly lead to excessive variables per page, which would fall on the development team to clean the extra and assign the correct shared ones. This goes against our Precision focus, and so we decided to drop this goal altogether.

IV. Preliminary Results

To evaluate our implementation, we measured the Precision and Recall statistics in 6 real past projects from OutSystems using their formulas (1, 2) respectively. Although these metrics are used on classification problems, we adapted their concepts into our work. Precision relates to the tool's ability to identify and instantiate a component correctly and Recall relates to how many components in the designs our tool covers.

$$Precision = \frac{truePositive}{truePositive + falsePositive} \tag{1}$$

$$Recall = \frac{truePositive}{truePositive + falseNegative} \tag{2}$$

To calculate the total number of components in each project, our tool already passes through every layer in the designs, so it was easily calculated. To count the True Positives, False Positives and False Negatives (True Negatives do not exist in this type of work) that are necessary to calculate the Precision and Recall statistics, these had to be manually counted. In our case, True Positives relate to how many components we are correctly identifying, False Positives are all the components we are identifying incorrectly and the False Negatives are all the other components that are calculated by subtracting the previous 2 from the total amount of components, since in our case True Negatives do not exist.

Table II summarizes our results. We achieved a global precision of 99.6% and recall of 80%. So, developers get a low number of misclassified components, mitigating the potential rework effort fixing wrongly create components.

Projects	Comp	TP	FP	FN	Precision	Recall
A	3310	2237	18	957	0.992	0.709
B	5876	4079	16	1057	0.997	0.820
C	1943	1378	2	382	0.999	0.803
D	4178	2994	17	713	0.995	0.829
E	569	434	5	102	0.989	0.819
F	1981	1634	5	266	0.997	0.865
Summary	17857	14317	63	3477	0.996	0.805

TABLE II
PRECISION AND RECALL FOR OUR 6 PROJECTS

At OutSystems there is a Customer Success department composed of 5 Front-End experts. This team is responsible for hand-crafting the Live Style Guide by translating the design screens and components into the OutSystems language. We sent this team our tool and the same six past projects to apply our tool and check the results by inspecting every single generated screen to compare the results of their work with the results obtained as the output of our tool. Since they are experts in this area and we are automating part of what they do, their feedback is the most important. These past projects were created without having our tool in mind and so results will reveal a significant variance. At the time of writing this paper, we got feedback from 2 of the 5 developers.

Our tool's effectiveness was worse when applied to designs where it was not only more challenging to identify the UI components correctly but also in those with a higher level of complexity. This complexity could be due to several reasons such as a higher presence of custom patterns or complex styling related techniques which we cannot detect. In these worst-case scenario projects, Developers would not use the result created by the tool as on average it would take more time than they usually have to implement the same set of pages originally delivered. To be clear, this adopt/not adopt the generated design is a fast decision to take.

For projects where our tool was more effective, the Front-End team can save up to 8h of the 16h they have (see table III). This is a 56% time reduction of the two workdays they usually have to invest in creating these pages. Developers noted that they expect they could deliver up to 400% more pages if they started working on top of the output, as can be seen in projects B and C, where they could deliver 6 more pages on top of the 2 they originally delivered.

The developers commented that after some more development and minor improvements, it could very well be a staple in their future workflow. These improvements were mostly focusing on styling issues (such as identifying styling classes and applying them to the components) which fall out of scope of this work.

Projects	Time to create the set (originally 16h)	Extra Pages (originally delivered)
A	28h	1 (2)
B	8h	6 (2)
C	16h	6 (2)
D	20h	0 (3)
E	10h	2 (3)
F	24h	0 (2)

TABLE III
DEVELOPER FEEDBACK FOR THE 6 PROJECTS

V. CONCLUSION

Our goal was to mitigate an existing gap in the collaboration between designers and front-end developers. This is vital for a great User Experience, which is a crucial factor in today's market and to cope with the current scarcity of available individuals with this expertise in the market. We implemented a solution that automates the generation of structured application screens in a Low-Code technology - work done manually today - by taking advantage of a design technology's structure and a library of Low-Code reusable UI components.

The results gathered from an OutSystems professional front-end development team suggest our tool may improve the value these teams can deliver to their customers. Depending on the complexity of the projects that the tool is being applied to, the team reported increases between 150%-400% in application screens created with a similar effort. In addition, by automating part of the composition process, we freed up time these professionals had to dedicate to this process and redirect most of their attention to topics requiring more knowledge and expertise. In the worst case scenario, where none of the generated screens are useful, the cost of applying the tool is nearly negligible. The developer can always choose to ignore the generated output and start the screens from scratch, as they normally would, costing them a couple of minutes assuming they were not working in parallel or running it in the background.

Our approach was based on unidirectional model transformations using an intermediate representation between our source and target models, which created a broader abstraction scope and allowed for an easier way to adapt to technologies different from those we used. Not only that, but the fact that this representation is a tree structure built by abstract nodes makes the tool easier to be evolved and extended to support more UI components.

The biggest challenge we encountered when implementing our solution was the heterogeneity of how UX and UI designers work. There are differences from one designer to another, but there are also notable inconsistencies in the approaches followed by a single designer for a particular project. These have severe implications on the tool's ability to accurately identify the used UI components, thus impacting the portion of designs the tool can convert automatically.

As future work, we will minimize the already light constraints we have in the design arrangement to be even less

intrusive in the design process. Another step is to extend this work to other design and front-end technologies. Since we mostly worked with JSON manipulation, another interesting investigation would be to apply a model transformation language to this approach. Finally, we plan to leverage the expertise of the tool users to increase recall while preserving precision in the transformation process.

Acknowledgment

This work has been funded by OutSystems Research and Development and is part of the collaboration between OutSystems and NOVA LINCS (UIDB/04516/2020).

We thank the OutSystems Advanced Development team and the Design team at Customer Success who provided insights about their practices and helped evaluate the tool.

References

[1] J. Johnson, *Internet users in the world 2020*, en, Jan. 2021. [Online]. Available: https://www.statista.com/statistics/617136/digital-population-worldwide/ (visited on 02/08/2021).

[2] U. Census, *Population Clock: World*, 2021. [Online]. Available: https://www.census.gov/popclock/world (visited on 02/08/2021).

[3] S. O'Dea, *Smartphone users 2020*, en, Dec. 2020. [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (visited on 02/08/2021).

[4] C. Candeias, *6 Steps to Improve User Experience: Embracing Your End-Users*, en, Jan. 2020. [Online]. Available: https://www.outsystems.com/blog/posts/improve-user-experience/ (visited on 07/02/2021).

[5] Oracle, *Seventy-Seven Percent Of Consumers Feel Inefficient Customer Service Experiences Detract From Their Quality of Life*, en, Apr. 2018. [Online]. Available: https://www.prnewswire.com/news-releases/seventy-seven-percent-of-consumers-feel-inefficient-customer-service-experiences-detract-from-their-quality-of-life-300626778.html (visited on 02/08/2021).

[6] T. Puthiyamadam and J. Reyes, *Experience is everything: Here's how to get it right*, en_us, Feb. 2017. [Online]. Available: https://www.pwc.com/us/en/zz-test/assets/pwc-consumer-intelligence-series-customer-experience.pdf (visited on 02/08/2021).

[7] D. Data, *New Research from Dimension Data Reveals Uncomfortable CX Truths*, en, Apr. 2017. [Online]. Available: https://www.prnewswire.com/news-releases/new-research-from-dimension-data-reveals-uncomfortable-cx-truths-300433878.html (visited on 02/08/2021).

[8] C. Pemberton, *Key Findings From the Customer Experience Survey*, en, Mar. 2018. [Online]. Available: https://www.gartner.com/en/marketing/insights/articles/key-findings-from-the-gartner-customer-experience-survey (visited on 07/02/2021).

[9] S. W. J. Kozlowski and D. R. Ilgen, "Enhancing the effectiveness of work groups and teams," *Psychological Science Suppl. S*, pp. 77–124, 2006.

[10] K. Kristensen and B. Kijl, "Collaborative Performance: Addressing the ROI of Collaboration," en, *International Journal of e-Collaboration*, vol. 6, no. 1, pp. 53–69, Jan. 2010, ISSN: 1548-3673, 1548-3681. DOI: 10.4018/jec.2010091104.

[11] S. Faraj and L. Sproull, "Coordinating Expertise in Software Development Teams," *Management Science*, vol. 46, pp. 1554–1568, Dec. 2000. DOI: 10.1287/mnsc.46.12.1554.12072.

[12] Y. Lindsjørn, D. I. K. Sjøberg, T. Dingsøyr, G. R. Bergersen, and T. Dybå, "Teamwork quality and project success in software development: A survey of agile development teams," en, *Journal of Systems and Software*, vol. 122, pp. 274–286, Dec. 2016, ISSN: 0164-1212. DOI: 10.1016/j.jss.2016.09.028.

[13] S. B.V, *The digital design toolkit*, en. [Online]. Available: https://www.sketch.com/ (visited on 02/18/2021).

[14] Figma, *Figma: The collaborative interface design tool.* en-US. [Online]. Available: https://www.figma.com/ (visited on 02/18/2021).

[15] Anima, *Anima — Design to development platform*. [Online]. Available: https://www.animaapp.com/ (visited on 02/10/2021).

[16] I. Inc, *Design-to-code — Design Defined — InVision*, en-US. [Online]. Available: https://www.invisionapp.com/design-defined/design-to-code/ (visited on 02/11/2021).

[17] T. Mens and P. Van Gorp, "A Taxonomy of Model Transformation," en, *Electronic Notes in Theoretical Computer Science*, Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), vol. 152, pp. 125–142, Mar. 2006, ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.10.021.

[18] F. Budinsky, M. Finnie, P. Yu, and J. Vlissides, "Automatic Code Generation from Design Patterns," *IBM Systems Journal*, pp. 151–171, 1996.

[19] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock, "Automatically generating personalized user interfaces with Supple," en, *Artificial Intelligence*, vol. 174, no. 12-13, pp. 910–950, Aug. 2010, ISSN: 00043702. DOI: 10.1016/j.artint.2010.05.005.

[20] J. Jelinek and P. Slavik, "GUI generation from annotated source code," in *Proceedings of the 3rd annual conference on Task models and diagrams*, ser. TAMODIA '04, New York, NY, USA: Association for Computing Machinery, Nov. 2004, pp. 129–136, ISBN: 978-1-59593-000-2. DOI: 10.1145/1045446.1045470.

[21] M. Macık, "Automatic user interface generation," PhD Thesis, PhD thesis, Faculty of Electrical Engineering, Czech Technical University, 2016.

[22] S. Studio, *Supernova*, en. [Online]. Available: https://supernova.io/ (visited on 02/10/2021).

[23] Yotako, *Seamless transition from design to code*. [Online]. Available: https://www.yotako.io/ (visited on 02/10/2021).

[24] N. Ltd, *React Studio*. [Online]. Available: https://reactstudio.com/ (visited on 02/10/2021).

[25] PixelCut, *PaintCode - Turn your drawings into Objective-C or Swift drawing code*, 2021. [Online]. Available: https://www.paintcodeapp.com/ (visited on 02/10/2021).

[26] Pagedraw, *Pagedraw/pagedraw*, original-date: 2019-01-12T20:13:39Z, Feb. 2021. [Online]. Available: https://github.com/Pagedraw/pagedraw (visited on 02/11/2021).

[27] M. Bexiga, "Closing the Gap Between Designers and Developers in a Low-Code Ecosystem," M.S. thesis, NOVA School of Science & Technology, 2021.

[28] Modulz, *Modulz*, en. [Online]. Available: https://modulz-website.now.sh/ (visited on 02/10/2021).

[29] T. Palmer, *2019 Design Tools Survey Results*, en. [Online]. Available: https://uxtools.co/survey-2019/ (visited on 02/16/2021).

[30] ——, *2020 Tools Survey Results*, en, Dec. 2020. [Online]. Available: https://uxtools.co/survey-2020/ (visited on 02/10/2021).

[31] PixelCut, *PaintCode Plugin for Sketch*, en, 2021. [Online]. Available: https://www.paintcodeapp.com/sketch (visited on 02/16/2021).

[32] K. Jiang, *Introducing: Figma to React*, en-US, Apr. 2018. [Online]. Available: https://www.figma.com/blog/introducing-figma-to-react/ (visited on 02/16/2021).

[33] M. Bexiga, S. Garbatov, and J. C. Seco, "Closing the gap between designers and developers in a low code ecosystem," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '20, New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1–10, ISBN: 978-1-4503-8135-2. DOI: 10.1145/3417990.3420195.

[34] M. Dixon and J. Fogarty, "Prefab: Implementing Advanced Behaviors Using Pixelbased Reverse Engineering of Interface Structure," in *In Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2010.

[35] T. Beltramelli, "Pix2code: Generating Code from a Graphical User Interface Screenshot," *arXiv:1705.07962 [cs]*, Sep. 2017, arXiv: 1705.07962. [Online]. Available: http://arxiv.org/abs/1705.07962 (visited on 01/18/2021).

[36] V. Jain, P. Agrawal, S. Banga, R. Kapoor, and S. Gulyani, "Sketch2Code: Transformation of Sketches to UI in Real-time Using Deep Neural Network," *arXiv:1910.08930 [cs, eess]*, Oct. 2019, arXiv: 1910.08930. [Online]. Available: http://arxiv.org/abs/1910.08930 (visited on 01/18/2021).

[37] S. Hassan, M. Arya, U. Bhardwaj, and S. Kole, "Extraction and Classification of User Interface Components from an Image," en, *International Journal of Pure and Applied Mathematics*, vol. 118, p. 16, ISSN: 1314-3395.

[38] T. A. Nguyen and C. Csallner, "Reverse Engineering Mobile Application User Interfaces with REMAUI (T)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 248–259. DOI: 10.1109/ASE.2015.32.

[39] J. Nielsen, *Paper Prototyping: Getting User Data Before You Code*, en. [Online]. Available: https://www.nngroup.com/articles/paper-prototyping/ (visited on 02/16/2021).

[40] J. M. Carroll, *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, en. Elsevier, May 2003, Google-Books-ID: gGyEOjkdpbYC, ISBN: 978-0-08-049141-7.

[41] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster, "UIML: An appliance-independent XML user interface language," en, *Computer Networks*, vol. 31, no. 11-16, pp. 1695–1708, May 1999, ISSN: 13891286. DOI: 10.1016/S1389-1286(99)00044-4.

[42] L. A. MacVittie, *XAML in a Nutshell*, en. "O'Reilly Media, Inc.", 2006, Google-Books-ID: v03elGOy9ogC, ISBN: 978-0-596-52673-3.

[43] K. C. Feldt, *Programming Firefox: Building Rich Internet Applications with XUL*, English, 1st edition. O'Reilly Media, Apr. 2007.

[44] B. Frost, *Atomic Design*, en. Brad Frost Web, Dec. 2016, Google-Books-ID: 1e92vgAACAAJ, ISBN: 978-0-9982966-0-9.

[45] S. B.V, *Layer Basics*, en. [Online]. Available: https://www.sketch.com/docs/layer-basics/ (visited on 02/16/2021).

[46] ——, *Symbols*, en. [Online]. Available: https://www.sketch.com/docs/symbols/ (visited on 02/18/2021).

[47] ——, *Shapes*, en. [Online]. Available: https://www.sketch.com/docs/shapes/ (visited on 02/18/2021).

[48] *Reuse and Refactor*, en, Jul. 2018. [Online]. Available: https://success.outsystems.com/Documentation/11/Developing_an_Application/Reuse_and_Refactor (visited on 02/09/2021).

[49] OutSystems, *Service Studio Overview*, en, Mar. 2019. [Online]. Available: https://success.outsystems.com/Documentation/11/Getting_started/Service_Studio_Overview (visited on 02/18/2021).

[50] ——, *Libraries*, en, Jul. 2019. [Online]. Available: https://success.outsystems.com/Documentation/11/Developing_an_Application/Reuse_and_Refactor/Libraries (visited on 02/18/2021).

[51] ——, *Extensions*, en, Jul. 2018. [Online]. Available: https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/Extend_Logic_with_Your_Own_Code/Extensions (visited on 02/18/2021).

[52] S. B.V, *File format*, en, 2021. [Online]. Available: https://developer.sketch.com/file-format/ (visited on 02/09/2021).