João Ricardo Alpoim Gonçalves

Bachelor in Computer Science

A model for widget composition in the OutSystems Platform

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: João Costa Seco, Assistant Professor,
Faculdade de Ciências e Tecnologias da
Universidade de Lisboa

Co-adviser: Hugo Lourenço, R&D Principal Software Engineer,
OutSystems

Examination Committee

Chairperson:	Fernanda Barbosa
Raporteur:	Alcides Miguel C. Aguiar Fonseca
Member:	João Ricardo Viegas da Costa Seco

December, 2019

A model for widget composition in the OutSystems Platform

Copyright © João Ricardo Alpoim Gonçalves, Faculty of Sciences and Technology, NOVA University Lisbon.
The Faculty of Sciences and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

I would like to thank Faculdade de Ciências e Tecnologias from the Universidade Nova de Lisboa for giving me the fundamental tools over the past five years that will now allow me to start my professional career. Thank you also to OutSystems for funding a scholarship for this dissertation and for allowing me to work beside them.

Thank you, Hugo Lourenço and João Costa Seco, for allowing me to enter on such a big project. A year has passed, but it felt much less. I will be forever grateful.

Thank you to my new friends: António Ferreira, David Mendes, Francisco Cunha, Francisco Gracias, Francisco Magalhães, Joana Tavares, Lara Borisoglebski, Mariana Cabeda, Michael Silva, Miguel Loureiro, Miguel Madeira, Nuno Calejo, Nuno Pulido, and a lot more people I got the chance to meet. If it hadn't been for you, this past year wouldn't have been so much fun.

Thank you, my best friends, for being the most fun people I have in my life.

The biggest thank you to my parents for sacrificing so much so that I got the chance to have a better life. Thank you, Hélder and Sara, who raised me and helped me whenever I asked them to. And last but not least, thank you, Carolina, for being with me when we had the time, hearing me grinding about writing when I just wanted to be with you.

This work is for all of you.

Abstract

Developers use visual programming languages for faster development of user interfaces due to better ease of use, readability, component reusability – widgets –, and an instant preview of the desired effects. However, the most common composition models to form user interfaces are *black-box*: combine existing widgets to form new widgets, but generally do not allow indiscriminate modification of their internal components.

The OutSystems platform provides a *What You See Is What You Get* (WYSIWYG) experience where developers can build user interfaces by assembling user interface elements from predefined building blocks: the more fundamental and native components (widgets) represent HTML elements, and custom-made building blocks (web blocks) represent reusable compositions. However, web blocks and widgets are not uniform. Currently, through some workarounds, creators can define compositions that, after instantiated, their inside components can be customizable by other developers, but they either do not follow OutSystems' good practices for creating web applications, do not show the user's customizations – no preview –, or need expertise that citizen developers do not have.

Our objectives with this work are to develop a new composition model for user interface components that allows to customize the properties of the inner elements of reusable compositions at the places where they are instantiated, integrate the model with the platform in a visual and interactive way where creators can control what can be modified, and users can customize respectively while getting a consistent preview.

Reusable compositions in the OutSystems language are unique and static. Thus, for developers to be able to change internal components of a composition and get a preview of that change, the underlying models must explicitly receive and transmit properties of the components internal state to the composition elements.

The work was validated by usability testing and by comparison between our solution with widgets that are specialized by OutSystems for specific use cases. The new presented approach is faster and more intuitive to use than what is currently offered by OutSystems. We also observed it works best in tandem with mechanisms already in place (e.g., input parameters) to offer more complete reusable compositions.

In the end, all objectives were met, providing a working solution which enables users to customize their or other's web blocks. With this work, reusable composition creators and users will get more control, customization possibilities, and user experience more intuitive, increasing productivity and user satisfaction.

Keywords: Low-code Development Platform, Visual Programming Languages, White box Composition Model, Component-Based Design, Component customization

Resumo

Os programadores usam linguagens de programação visual para um desenvolvimento mais rápido das interfaces de utilizador devido à maior facilidade de uso, legibilidade, reutilização de componentes – widgets – e uma visualização instantânea dos efeitos desejados. No entanto, os modelos de composição mais comuns para formar interfaces de utilizador são *black-box*: combinam os widgets existentes para formar novos widgets, mas geralmente não permitem modificações indiscriminadas dos seus componentes internos.

A plataforma OutSystems fornece uma experiência *What You See Is What You Get* (WYSIWYG), na qual os programadores podem criar interfaces de utilizador ao montar elementos da interface de utilizador a partir de blocos de construção predefinidos: os componentes mais básicos e nativos (widgets) representam elementos HTML e blocos de construção personalizados (web blocks) representam composições reutilizáveis. No entanto, web blocks e widgets não são uniformes. Atualmente, por meio de soluções alternativas, os criadores podem definir composições que, após instanciadas, os componentes internos podem ser customizados por outros programadores, mas estas não seguem as boas práticas da OutSystems para criar aplicações Web, não mostram as customizações do utilizador – sem preview –, ou são necessários conhecimentos que os programadores podem não possuem.

Os nossos objetivos com este trabalho são desenvolver um novo modelo de composição para componentes da interfaces de utilizador que permita customizar as propriedades dos elementos internos das composições reutilizáveis nos locais em que são instanciadas, integrar o modelo à plataforma de maneira visual e interativa, onde os criadores podem controlar o que pode ser modificado e os utilizadores podem customizar respectivamente enquanto obtêm uma visualização consistente.

As composições reutilizáveis na linguagem OutSystems são únicas e estáticas. Assim, para que os programadores possam alterar os componentes internos de uma composição e obter um *preview* dessa alteração, os modelos subjacentes devem receber e transmitir explicitamente propriedades do estado interno dos componentes para os elementos da composição.

O trabalho foi validado através de testes de usabilidade e comparação entre a nossa solução e com widgets especializados pela OutSystems para casos de uso específicos. A nova abordagem apresentada é mais rápida e mais intuitiva para usar do que o modelo de composição que é atualmente oferecido pela OutSystems. Também observámos que é mais eficiente usar em conjunto com os mecanismos já existentes (e.g., parâmetros de entrada) para oferecer composições reutilizáveis mais completas.

No final, todos os objetivos foram alcançados, fornecendo uma solução funcional que permite aos utilizadores customizar os seus web blocks ou os de outros. Com este trabalho,

criadores e utilizadores de composições reutilizáveis terão mais controlo, possibilidades de customização e experiência do utilizador mais intuitiva, aumentando a produtividade e a satisfação do programador.

Palavras-chave: Plataforma de Desenvolvimento *Low-code*, Linguagens de Programação Visual, Modelo de Composição *White-box*, Design baseado em Componentes, Customização de componentes

Contents

List of Figures xv

List of Tables xvii

Listings xix

Glossary xxi

Acronyms xxiii

1 Introduction 1

1.1 Context 1
1.2 Motivation 4
1.3 Objectives 4
1.4 Contributions 6
1.5 Structure 6

2 Background 7

2.1 Component-Oriented Programming 7
2.1.1 Motivation 8
2.1.2 Component Characteristics 8
2.1.3 Comparison with OOP 10
2.2 Composition 10
2.2.1 Views 11
2.2.2 Categories 11

3 Compositions in OutSystems 13

3.1 OutSystems Platform 13
3.1.1 Service Studio 13
3.1.2 Platform Server 14
3.2 Workspace 15
3.2.1 Actions 15
3.2.2 Screens 15
3.2.3 Widgets 16

3.2.4 Web Blocks . . . 16
3.3 Web Block Customization . . . 17
3.3.1 Input Parameters . . . 18
3.3.2 CSS Styling . . . 19
3.3.3 Duplication . . . 20
3.3.4 Comparison between Workarounds . . . 20

4 Related work **23**
4.1 Composition Models . . . 23
4.1.1 *UI* Families . . . 24
4.1.2 Customization by input . . . 25
4.2 Composition in Visual Languages . . . 25

5 New Composition model **27**
5.1 Current model . . . 27
5.1.1 Widgets . . . 28
5.1.2 Web Blocks . . . 29
5.1.3 Web Block Instances . . . 30
5.2 Enabling customization . . . 31
5.2.1 Requirements . . . 31
5.2.2 New Model . . . 32
5.3 Model comparison . . . 33
5.4 Summary . . . 37

6 Implementation **39**
6.1 Architecture . . . 40
6.1.1 Model . . . 41
6.1.2 View . . . 44
6.1.3 Presenter . . . 45
6.2 Compiler . . . 50

7 Evaluation **59**
7.1 Usability Testing . . . 59
7.2 Solution comparison . . . 62
7.2.1 Analysis methodology . . . 62
7.2.2 OutSystemsUI . . . 63
7.2.3 Examples . . . 63
7.2.4 Comparison . . . 64

8 Conclusions **65**

Bibliography **69**

A Proof of concept 71

B Usability Testing: Script 75

C Usability Testing: Results 79

D Solution Comparison: Results 81

I Short article 83

List of Figures

1.1 Widget selected in a web block's definition. 2
1.2 Web block's instance on a screen. 3

3.1 Service Studio sends the application to the Platform Server to be compiled and deployed [12]. 14
3.2 Service Studio's Workplace 15
3.3 Widget tree of a web screen 16
3.4 Widget toolbox. 17
3.5 Input widget's exposed properties. 17
3.6 Button widget's exposed properties. 17
3.7 Web block instance with custom parameters. 18
3.8 Input widget inside the Web block definition with parameters bound to it. . 19
3.9 Screen's style sheet with customized selector (red box). 20

4.1 Differences between a building block's definition and corresponding instance 26

5.1 Model of OutSystems' building blocks. 28
5.2 New structures added to support web block customization. 32
5.3 Web block for requesting a user's address - Address web block. 33

6.1 *MVC* design pattern. 40
6.2 *MVP* design pattern. 40
6.3 Time line of when a widget is changed. 41
6.4 The command turns the widget *Input_Name* properties customizable. 46
6.5 Properties Editor with *Input_ZipCode* selected. 49
6.6 How the preview is drawn by the View. 51
6.7 How the compiled application draws its web blocks. 51
6.8 Order of compilation for the components of an application model. 52

7.1 Average and standard deviation of SUS for each group (higher is better). . . . 61
7.2 Average and standard deviation of time taken to complete each group's task (lower is better). 61

D.1 OutSystemsUI widgets results. 82

List of Tables

2.1 Comparison between structured programming (SP), object oriented programming (OOP) and component oriented programming (COP) [23]. 10

3.1 Approaches comparison. Best ones are the ones that have the most check marks. 21

7.1 Average SUS and time taken to complete the task for each group. 60

C.1 Beta group results. 80
C.2 Alfa group results. 80
C.3 System Usability Scale questions. 80

D.1 Indexes abbreviations. 81

Listings

1.1 Improved composition model 5
4.1 Bean example [23]. 24
5.1 Simplified widget model instanced for the "Input.ZipCode"widget in XML. 29
5.2 Input widget HTML template. 29
5.3 Simplified web block model instanced for the web block "Address"in XML. 30
5.4 Simplified web block instance model for the instance of the web block "Address"in XML. 30
5.5 Address Web Block's child widgets in the current model. 34
5.6 Address Web Block's child widgets in the new model. 34
5.7 Address Web Block instance in the current model. 35
5.8 Address Web Block instance in the new model. 36
6.1 New structures added to ObjectDefinitions. 42
6.2 New web block class definition. 43
6.3 New web block instance class definition. 44
6.4 Screen in Figure 5.3 from the *UPPS* app compiled. 53
6.5 Web block *Address* from the *UPPS* app compiled. 54
6.6 The new formatted property value. 56
6.7 The new compiled property. 56
6.8 Address component in Listing 6.5 after adaptation. 58
6.9 Screen component in Listing 6.4 after adaptation. 58
A.1 The component *Address* with the hook *zipCode* which is instantiated with the parameter passed in the state. 71
A.2 The component *Address* adapted to the new model. 72
A.3 The component *App* that instantiates the component *Address* with the *state*. 73
A.4 The component *App* adapted to the new model. 73

Glossary

action	Logic that runs either on server or client side and can be either Preparation or Data actions, depending on the type of app - web or mobile, respectively, client or server actions .
black-box	Device, system or object which can be viewed in terms of its inputs and outputs (or transfer characteristics), without any knowledge of its internal workings.
component	Web resource that encapsulates a set of related functions and data.
Component-Based Design	Development approach based on reusability that composes systems with loosely independent components.
web block	Developer made reusable screen composed with widgets or web blocks.
white-box	Contrary to black-boxes, these expose everything to the outside scope.
widget	A unitary screen element that can be either a HTML element or a reusable component (web block).

Acronyms

COP	Component-Oriented Programming.
DSL	Domain-Specific Language.
GPPL	General-Purpose Programming Language.
GUI	Graphical User Interface.
IDE	Integrated Development Environment.
LCDP	Low-Code Development Platform.
OOP	Object-Oriented Programming.
PL	Programming Language.
UI	User Interface.
VPL	Visual Programming Language.
WYSIWYG	What You See Is What You Get.

Chapter 1

Introduction

Developers can create applications faster with Low-Code Development Platforms that replace textual code environments with easy to learn Graphical User Interfaces (GUIs) abstractions and reusable components already implemented and fully tested before release. In Component-Based Design, components encapsulate related functions and data, restricting access to their details (abstraction), and composition mechanisms, that may compromise the passing of information necessary to inspect and change some aspects of the components.

However, Low-Code Development Platforms (LCDPs) require some degree of parametricity of components to fit every possible use case of component instantiation and configuration. Since component mechanisms are usually black-box, we seek a new form of composition to be applied to UI components in the OutSystems Platform.

1.1 Context

An LCDP is an environment for application development that relies on declarative and visual methods with minimal textual and manual coding. It offers software development companies many advantages over traditional alternatives (e.g., reduced amount of textual and manual coding and more emphasis on business logic), resulting in faster development. Citizen, business and pro developers can cooperate in this way as there's no longer the restriction that only skilled programmers can implement feature-rich applications [16].

For citizen developers, LCDPs provide easy to learn visual languages with pre-defined native components that replace many web resources (e.g., HTML elements) with visual elements, granting more time for developers to define user interfaces for all kinds of devices, data models, business logic and workflow. Therefore, development in a visual

language that does not support customization of all components limits the user experience, because component implementation is encapsulated in visual elements with a limited interface. Although, systems are developed in a DSL, mainstream tecnologies are used to produce the actual systems's code and deploy it.

In a Component-Based Design, components encapsulate their implementation, revealing no details to the outside scope (black-box) [4], connecting to other components by interfaces. However, for components that have a visual representation, customizable details should be visible from the outside. A way of having a disciplined and consistent way of manipulating componets' details is through a novel composition mechanism that encapsulates and parameterizes one component as a functional abstraction [11], and a visual editor ready for the added information exchange.

Even though LCDP for front-end web that provide many components that replace current web resources (e.g., HTML elements), it is not wise to deliver every possible scenario, but rather giving the user the ability to create their own. In the front-end development, screens compose the User Interface (UI) with which end-users will interact with visual components. If a part of a screen can be used on more screens, then users should be able to create a reusable component out of it and use it on any other application. Composition of components means combining two or more components to create a more complex one, merging the best of both worlds: component's reusability with screen's composability.

Low-code environments for front-end web development provide customization tools for the end-user UI, supported by screen editors where developers compose screens by dragging and dropping visual elements. OutSystems delivers an editor where the developer can customize the components inside screens. Service Studio[1] contains many widgets (OutSystems' visual native components) that represent many of the existent web resources today and allows the definition of web blocks, that are reusable screen parts.

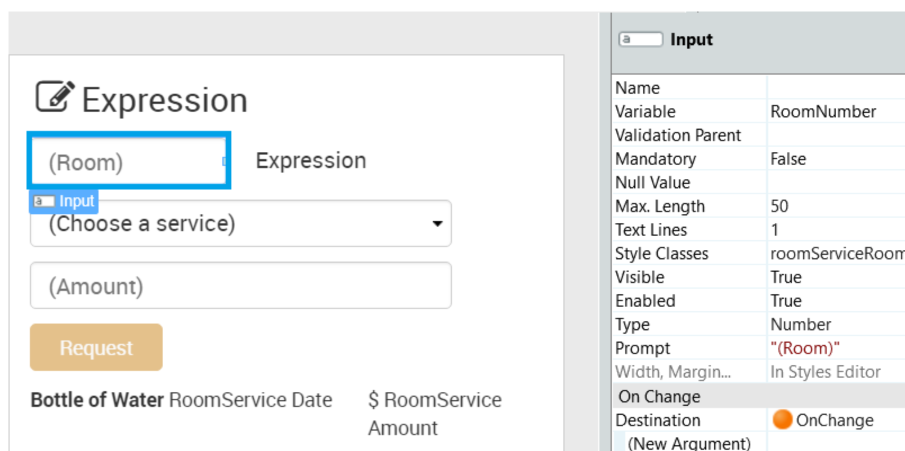


Figure 1.1: Widget selected in a web block's definition.

[1]OutSystems development environment.

In Figure 1.1, while defining a web block, a user can access every property of the selected (small blue box) widget in the list of parameters passed to the block on the properties editor on the right of Figure 1.1, which is a visual representation of an Input HTML element.

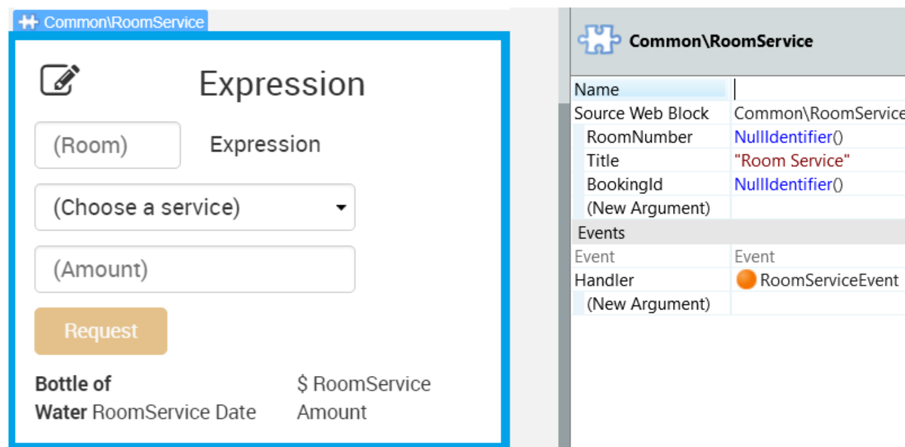


Figure 1.2: Web block's instance on a screen.

In Figure 1.2, when the user tries to select inside the block, the platform can only select the block as a whole (big blue box). The block becomes a black-box: developers have no access to the inside scope, and the only explicit parameters of the internal representation are available (RoomNumber, Title and BookingId in the properties editor of the block in Figure 1.2). Web blocks were created to replace a repeated pattern used on other screens, but as customization was not affected while using normal screens, using web blocks compromise customization, due to a composition model that does not respect the platform's consistency.

The current version of Service Studio supports customization for widgets, but web blocks lack in ease of use. When trying to customize an instance of a web block, the platform disables all access to widgets inside (Figure 1.2). There are ways to do it, but they either lack a preview feature, which is a big letdown for a visual editor that should show in real time the changes being made, require knowledge of complex languages, are hard to maintain or duplicate code.

Both in text-based and visual programming languages, component customization is limited by how the creator predicted the component would be used and the language used (text-based Programming Languages (PLs) have encapsulation, visual may not have). In the OutSystems platform, web blocks aren't black-boxes because in design-time the user can define them and get a consistent visual representation, but when instantiating one on a screen, the developer loses this capability, breaking the user expectations to customize a block wherever it is.

Thus, the problem in which we will try to solve in this thesis is how to make a visual language capable of safely customizing component compositions without losing platform's consistency.

1.2 Motivation

Visual UI components with black-box composition raises difficulties for developers to customize instantiated compositions. Workarounds exist, but none follows a white-box model that is easy to configure. Thus, a composition model with a white-box reuse paradigm provides developers the fastest development experience.

Consider the following message exchange between two people on an OutSystems' forum discussion:

"How to chanage web block background color?? Help me resolve this. i inned to change web block bg color [sic]"

"(...) if you put a container around your content inside your web block, you can change the background color applying CSS to this container, using the Style Editor. What exactly is blocking you?"

For an LCDP, the solution given by the second person should not be the right one, but instead only having to select the top container of the web block and change the parameter responsible for the color inside the styles editor should be enough. Even though the solution works, it is not intuitive and breaks the platform's consistency, resulting in frustration. Consequently, the user could opt for duplicated code, compromising reusability.

Thus, in this work's context, reusable user-defined components are a must have because using other people's components is faster than implementing them from scratch. Real-time preview is also a very important requirement as interaction on a visual language should be responsive, otherwise, developers cannot have feedback of their changes.

1.3 Objectives

The objectives of this work are to revise and improve the existent composition of user interface elements in the OutSystems platform, with the same standards of any other component (reusability, instant preview). Therefore, the key objective is to make component compositions into white-boxes (access to every component's instance inside).

We will develop a component model based on the syntax presented in Listing 1.1 for the representation of web resources. So instead of delegating only the mandatory information to components' explicitly exposed properties so these can be rendered, they will have a rich interface to change its instance just like changing its definition. In this process, it is important to define what component properties can be exposed in the composition model's interface, even though, as default, it is expected to expose all properties

responsible for appearance and component behavior, so that we have a uniform model for every component.

Our next objective is to extend the OutSystems Domain-Specific Language (DSL) to be compatible with the new revised model, and then adapting the compiler that translates the model into mainstream languages and frameworks (TypeScript and React).

```
Component2Props {
   name: string
   color: string
}

Component 2 (c2p){
   public   property props:Component2Props = {
      name = if c2p.name is null then "Default hello" else c2p.name
      color = if c2p.color is null then "gray" else c2p.name
      ...
   };
   ...
   render() {...}
}

Component 1 {
   public   property name:string;
   private  property color:string; //private as so defined by the creator
   render() {
      ...
      Component 2 as c2 with {
         c2p of type Component2Props with {
            name  = this.name,
            color = red
         }
      }
   }
}

render Component 1 with {
   name = "Hello World"
}
```

Listing 1.1: Improved composition model

The solution should provide a way to customize compositions like a developer edits a component, delivering each with their own definition to the outside: every aspect customizable made possible by the creator.

1.4 Contributions

This work contributes with new ways of visual element composition that in a composition mechanism can provide a visual representation loyal to their implementation, delivering an intuitive and satisfactory experience to the user.

In the end, we provide a working prototype capable of being easier and faster to use than current solutions provided by the OutSystems platform and also be integrated into the environment. From the creator to the user, we reinvented the whole process that involves reusable compositions. Our solution also completes the user experience, since many of the mechanisms already implemented by OutSystems help developers to create their applications faster.

Finally, the adapted compiler allows to connect the final link between designing application and deploying them.

1.5 Structure

The work has the upcoming order of sections:

- Chapter 2 - Background: Research of the platform and knowledge about Component-Based Design, Composition Mechanisms and Models, and Visual Programming Languages;
- Chapter 3 - Compositions in OutSystems: Introduction to the OutSystems platform, definitions related to this work and solutions for this thesis problem;
- Chapter 4 - Related work: In this chapter we presented literature review of works that address this issue;
- Chapter 5 - New Composition model: After discovering the problem, we designed a new composition model, which we discuss in here;
- Chapter 6 - Implementation: After designing came the implementation, where we started by changing the OutSystems platform architecture, and then the compiler;
- Chapter 7 - Evaluation: To validate our work and attest that our solution offered improvements, we gathered a group sample to follow a simple task and compared their results when using our approach and what is already provided by the platform. We also compare our solution with some specialized widgets provided by the platform.

Chapter 2

Background

Before discussing any solution to the problem, we explore the background crucial to its understanding. In this Chapter, we will discuss component-oriented programming, composition mechanisms both in programming languages and frameworks, visual languages for UI and solutions for composing reusable components for a visual language.

2.1 Component-Oriented Programming

The "Majority of software engineers prefer to build software products from the ground up"[9], using for their assistance frameworks, design patterns and programming idioms. Component-Oriented Programming (COP) is a programming paradigm for software development [23], with a divide and conquer[1] approach that reduces as much as possible the development effort by encouraging the reuse of pre-built components (subproblems) to create applications (problem).

As the name suggests, components are parts of a whole (system). In a software context, components can be attached or not to a system [23], because they are designed to be so, as these represent higher level abstractions of self-contained, highly coupled functions [11] and can be assembled with other components through their interfaces [23]. Thus, systems in Component-Based Design are compositions of components.

Compared to other approaches, COP offers higher reusability and better modular structure with greater flexibility. By usign interfaces, programmers no longer need to know how components are implemented, as long interfaces remain unchanged, changes in interface implementations do not affect the soundness of programs [23].

[1] Breaks down a problem into subproblems, smaller in size, and combines the solutions of these subproblems to come up with a solution for the original problem [2].

2.1.1 Motivation

Since the Industrial Revolution, businesses "have adopted component standards for interchangeable parts and streamlined assembly tools"[23] for a faster development of complex products, replacing skilled with unskilled workers. In the software industry, however, products are still mainly manually made, resulting in tendencies of low-productivity, low-quality and overrun projects [23].

Creating quality software is a key issue of software engineering, even more for Platforms as a Service (PaaS) (e.g., OutSystems) because they are liable for any faults that may happen after deployment. Components are a way to create quality software because they are independent of context [23]: if a fault exists, just fix the problem and deploy, instead of worrying about changing accidentally other software or running entire systems just to test the proposed correction.

According to An Wang et al. [23], Component-Based Design has three major goals: conquering complexity, managing change, and reuse.

- Conquering complexity: The size and complexity of software is increasing and Component-Based Design deals with this with a divide and conquer approach.

- Managing Change: Software is always changing, and at every moment a programmer can deploy a new version with a fault if they are not careful. Components are best suited for constant software changes because they are easy to adapt. Software engineers agree that the best way for managing changes is to build systems out of reusable components.

- Reuse: Software reuse decreases development time and has increased quality because design and implementation are done only once (may suffer consequent updates). Reuse also means software can be used many times in different contexts, which allows large productivity gains. Component-Based Design compared to other solutions (e.g. Class libraries, Object-Oriented Programming Languages) delivers the highest level of software reuse because it supports more types of reuse (white-box, black-box, gray-box) that help determine if components can be modified.

Thus, Component-Based Design is the best solution for large software systems as it ensures a "manageable solution to deal with the complexity of software, the constant change of systems, and the problems of software reuse"[23].

2.1.2 Component Characteristics

A software component is a piece of self-contained, self-deployable computer code with well-defined functionality and can be assembled with other components through its interface [23]. Manickam et al. [9] define components with the following characteristics:

- Part of a whole: the most fundamental aspect of components is they are the product of decomposition of a system. Their composition should result in a complete functional system. According to Sametinger [19], "components describe and/or perform specific functions". Thus, part of a whole can also be compatible with Sametinger's definition of functionality.
- Component Ecosystem: Sametinger defines components as self-contained: reusable without the need to use other components. Manickam et al. notes "components are designed to be used in a compositional way together with other components"(not in isolation). Although components are meant to be independent of other components to be reusable, their composition in a "framework governed by a component model"form a component ecosystem.
- Component Framework: frameworks provide environments where components may be composed, because they have "well-defined interfaces that establish the protocols for component cooperation within the framework" [24].
- Component Model: defines what components represent, how they are built, how they can be composed and deployed. Frameworks use component models to establish protocols necessary for component compatibility.
- Component Interfaces: components can specify one or more interfaces with which other components can use for composition and, in turn, interfaces follow the composition standards established in the component model so that components can compose with others.
- Provided and required Interfaces: components can have two types of interface. Interfaces for other components to use are *provided interfaces*. Interfaces which other components use to compose with and add functionality are *required interfaces*. A component's interface would include the two types of interface to specify it.
- Component Compatibility: if one component's provided interface is compatible with another's required interface, then they are compatible and can be composed together.
- Implementation Independence: Changes to a component's implementation should not interfere with its compositionality: two components with compatible interfaces will always be composable if one is because the composition operation only uses their interfaces.
- Producer-Consumer Independence: as long as the producer of components and its consumer have a common interface definition for it, then components can be exchanged between the two. Sametinger also defines documentation to be of importance, as "The most useful component is rendered useless (...) when appropriate documentation is not available".

Capabilities	SP	OOP	COP
Divide and Conquer	X	X	X
Unification of Data and Function		X	X
Identity		X	X
Encapsulation			X
Deployment			X

Table 2.1: Comparison between structured programming (SP), object oriented programming (OOP) and component oriented programming (COP) [23].

2.1.3 Comparison with OOP

In Object-Oriented Programming, objects are the basic programming elements to encapsulate data and behavior [23]. Much like components, objects are used to breaks problems into manageable pieces (divide and conquer). Classes are the objects specifications. Both provide an abstraction ideal for reuse. But, components surpass objects in many ways, as we will demonstrate.

In Table 2.1, Wang [23] presents a summary between structured programming [2] (SP), with a low level of reuse; OOP, a high level of reuse; and COP with the highest level. Structured programming was the first to break a large problem into smaller ones, but lacks in every other category: unification of data and function allows for better cohesion between logic that is highly coupled; encapsulation enables abstraction of concepts – client does not need to know what the software does –; identity allows for each software entity to have a unique identity.

Components are self-deployable because they can be installed and executed in any end user's environment. Objects are not, since they are bound to OO languages. Object-Oriented Programming (OOP) supports interfaces, but, according to Wang, these don't have a clear relationship among superclasses and subclasses.

But what differentiates the most the two approaches is the coupling between elements: COP offers loosely coupled components, and OOP tightly coupled objects because of inheritance. Low coupling ensures self-containedness (independent of other constructs to be deployed). Thus, COP has the highest level reuse paradigm.

2.2 Composition

Software composition [10] means combining software constructs of a particular problem domain to create software applications [6]. Composition is the key factor in a Component-Based Design because it is the interaction between units of composition - software units that define behavior - and through standard mechanisms, larger units of behavior can be built [6].

[2] Lowest level of a divide and conquer approach, a precursor to OOP [3].

In the coming sections, we will introduce how systems are separated into components through every existing manner (view) of accomplishing software composition, every mechanism that follows a specific view and a summary of which should be a good composition mechanism for our problem.

2.2.1 Views

Views of software composition determine what units of composition are meaningful. According to Kung-Liu and Tauseef [6], there are three views in which software composition is modeled:

- **Programming View**: In this view, "any programming language construct is a unit of composition". To combine these is with another construct of the programming language. It is the smallest level of a composition and overlooks many of the problems a Component-Based Design thrives to solve (e.g. software reuse).

- **Construction View**: According to Nierstrasz et al. [11], software composition "is the process of constructing applications by interconnecting software components through their plugs"; presenting the next view. Here composition is at an higher-level of abstraction than the Programming View, since units of composition are pre-existing program units that connect each other through scripting languages (e.g., glue). Although, reuse is perceived as possible, the view does not assume components are external to the system development, and neither does the software architecture of the system mentions their reuse.

- **Component-Based Development (CBD) View**: In the CBD view, components have a model that defines their syntax and semantics; and how they can be composed. Thus, the model must follow a composition standard so that it can be highly composable. It is very similar to the previous view, but in here components are assumed to be from third-parties.

2.2.2 Categories

The views presented before relate to how components can be designed, but composition mechanisms can be further categorized into four groups:

- **Containment**: refers to nested definition; container unit's behavior is defined from their contained units. Composition model of several widgets is an example of containment;

- **Extension**: the behavior of a unit extends behaviors of other units. Multiple inheritance is an example of this;

- **Connection**: a behavior defined through the interaction of multiple units. Thus, Connection is message passing. This includes object delegation, where objects that send each other messages have a tight coupling;
- **Coordination**: defines a unit's behavior that coordinates the behaviors of multiples units. The difference with connection is that units do not communicate between themselves and only with the coordinator, removing all the coupling. Examples focus in data coordination and not in a composition model for nested units, so we will not pursue this way of thinking.

Chapter 3

Compositions in OutSystems

OutSystems is a founder of the Low-Code Development Platforms market [16], delivering visual models that rely on a higher abstraction language easy to learn. Helps users to develop without the worry of writing bad code and to create fast, secure, reliable, highly available and scalable web and mobile applications.

In this Chapter, we will introduce the platform, relevant definitions and explain how compositions of components are implemented.

3.1 OutSystems Platform

The *OutSystems* platform covers every step of application lifecycle management process with development and deployment environments and management consoles [13]. The user interacts with these to create their applications in OutSystems language that end up being compiled and deployed on the platform server as demonstrated in Figure 3.1.

3.1.1 Service Studio

Service Studio is the Integrated Development Environment (IDE) where users create all the layers of the application mostly by dragging and dropping visual elements. These layers are: the data model (what will be stored), application logic (how the application will behave), UI (what the end-product will look like to the end-user), business process flows (representations of business processes), integrations (components from outside the environment integrated into the application), security policies (restrict access to certain pages to certain people). Applications can consume and expose also SOAP and REST web services.

Wrong usage can evolve into errors, even on visual languages (e.g., missing information required to deploy correctly the application), and for this Service Studio uses

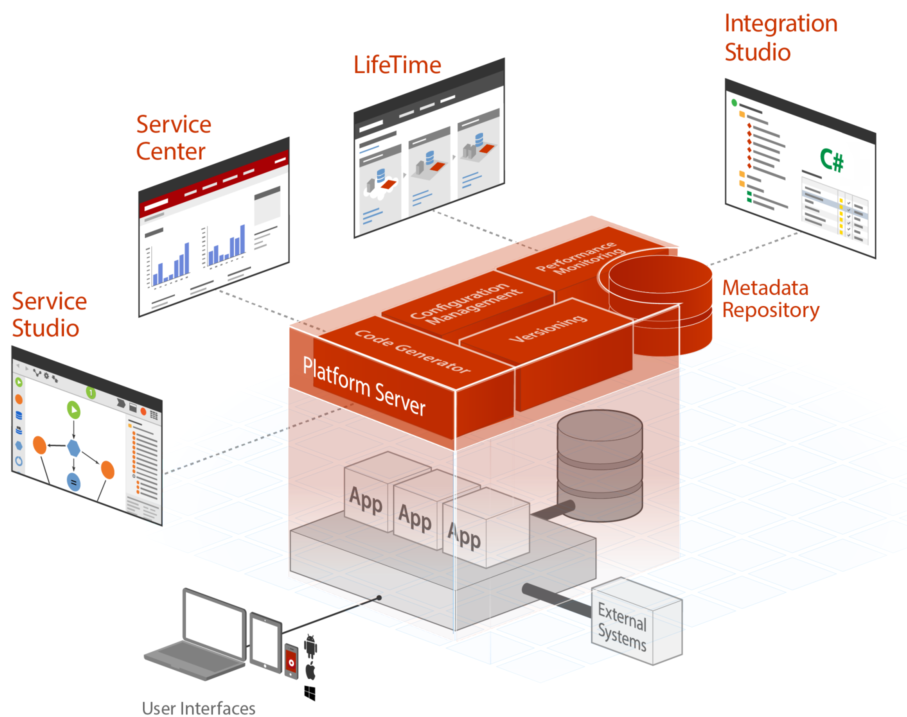


Figure 3.1: Service Studio sends the application to the Platform Server to be compiled and deployed [12].

a full self-healing and reference-checking engine to catch errors and changes that can impact existing applications, alerting the developer if it catches one. To ensure performance, the environment displaces work from the user, sending a compressed version of the application to the Platform Server [13].

3.1.2 Platform Server

This server component handles every task related to the compilation and deployment of applications. It's the backbone of the OutSystems platform. It generates, optimizes, compiles and deploys an OutSystems application to a standard web application server.

Before compilation starts, Service Studio validates the application model (first error caught stops the deployment) and sends it to the Platform Server. In the Code Generator, the application will be versioned, compiled to native .NET, allowing generated applications optimized for performance, secure and run on top of standard technologies; and deploy to all front-end servers in the developer's server farm (where developer's applications are stored) [13].

3.2 Workspace

Modules in Service Studio allow to structure applications into pieces with different purposes. The editor area has a number of different containers that help editing the screen as seen on fig. 3.2.

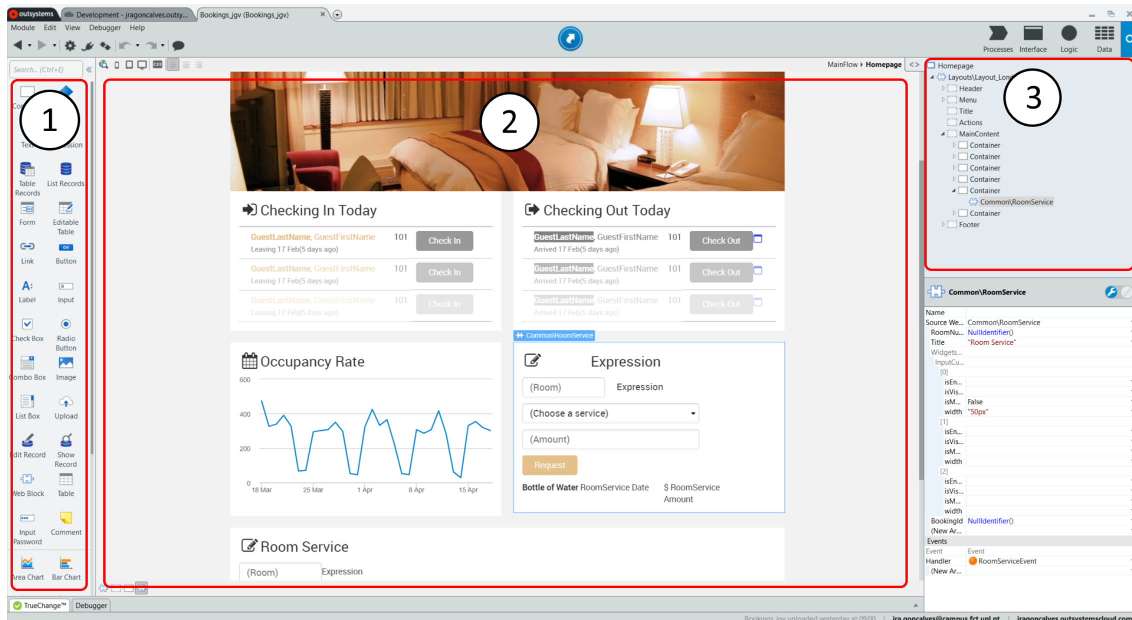


Figure 3.2: Service Studio's Workplace
1: Widget Toolbox 2: Canvas 3: Widget tree

The canvas, or the What You See Is What You Get (WYSIWYG) editor, is where the preview of the module happens and in real-time shows the aspect of a web screen. In here, it is shown how widgets are to be rendered when deploying, as one can add or remove elements and select each presented on the screen to edit further on adjacent editors.

On the right, the tree area can either show the interface of the module (application logic) or the composition of a screen. Below this resides the properties area when a screen or a widget is selected, where the user can define the parameters of it.

3.2.1 Actions

Actions are business logic that a user can create to be activated by the end-user for when accessing a screen (preparation) or interacting with a screen (client actions). Actions that execute on server (server actions) are defined in the Logic tab and not the Interface, as these are to be called from any screens' preparation or client actions.

3.2.2 Screens

Screens are the UIs that end-users interact with. They can be composed of widgets or web blocks. This composition is a tree of elements (Figure 3.3), resembling the structure seen

on a markup language, which can be changed by dragging components inside or outside of other component scopes. They can also have their own actions, input parameters and local variables.

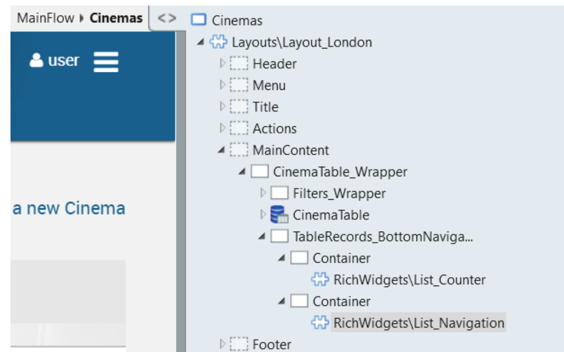


Figure 3.3: Widget tree of a web screen

3.2.3 Widgets

Widgets are the building blocks of the platform to create the UI. Each one represents elements of the UI presented to the end-users that display information and/or are interacted with.

The widget toolbox (Figure 3.4) houses every native building block provided by OutSystems. Despite complex building blocks (e.g., charts) or even web blocks residing here, they are all considered widgets. Thus, to create a form in a screen, the user drags the Form widget to the canvas and fills the information required for it to function. The Input widget allows users to extend forms even further with one more space for end-users to fill. It also has its properties (Figure 3.5). Mandatory properties are highlighted in red, and, in Figure 3.5, *Variable* needs to be filled, because it is where the widget will store the value entered by the user and send on a request.

Buttons may also be added to forms. Buttons widgets have their own properties (Figure 3.6), with *Destination* being highlighted because it is necessary to know what does that button do (trigger actions or navigate to a screen).

3.2.4 Web Blocks

To create screen parts that are reusable (can be used more than once for other purposes) web blocks are the answer. This means that a web block can have its own actions, logic, input and output parameters, and variables. Web blocks encapsulate multiple widgets, which means they can be either native widgets or other web blocks.

In Figure 1.2, the blue box around the web block is the most a developer can select of a web block. Web blocks' properties are *Name* (unique name that helps identifying the web block), the *Source Web Block* (which web block is the instance based on), the ones

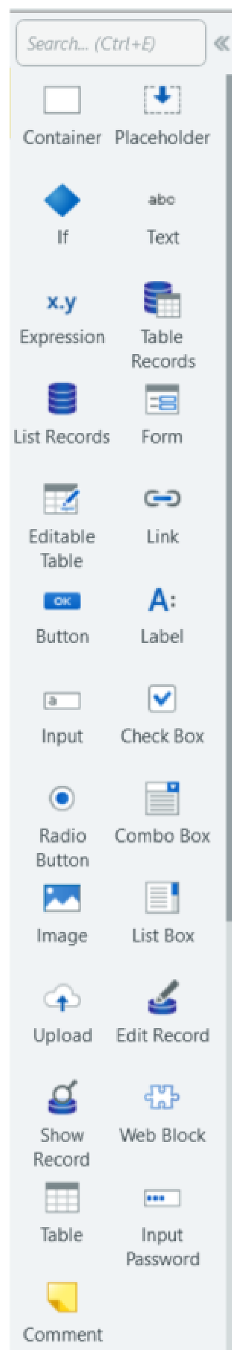


Figure 3.4: Widget toolbox.

Input	
Name	
Variable	
Validation Parent	
Mandatory	False
Null Value	
Max. Length	50
Text Lines	1
Style Classes	input
Visible	True
Enabled	True
Type	Text
Prompt	
Width, Margin...	In Styles Editor
On Change	
Destination	
Extended Properties	
Property	
= Value	

Figure 3.5: Input widget's exposed properties.

OK Button	
Name	
Label	"Ok"
Example	
Validation Parent	
Style Classes	Button
Visible	True
Enabled	True
Is Default	No
Width, Margin...	In Styles Editor
On Click	
Method	Submit
Validation	Server
Confirmation Mess...	
Destination	
Extended Properties	
Property	
= Value	

Figure 3.6: Button widget's exposed properties.

defined by the developer and events (logic triggered by the inside widgets to execute a handler outside the web block's scope).

3.3 Web Block Customization

Native widgets allow users access to all relevant properties to customize the visual representations, just like in a text-based programming language (Figure 1.1). Customizing

inside a normal screen or in a web block is the same: users can access the same properties in both situations.

In an application with several instances of the same web block, users may want to customize differently each according to their surroundings. However, as seen in Figure 1.2, access to the inside components is not possible because it is restricted by the platform.

Some workarounds exist to tackle this issue and in the next subsections we will present some of them, but as we will see, they are undesirable because they increase application complexity and are not easy to use for every developer.

3.3.1 Input Parameters

Like any other screen, web blocks can have multiple input parameters. So, a developer creating a web block can use input parameters to expose properties of the widgets. However, this is only possible for properties that are expressions, and, thus, evaluated at runtime. In Figure 3.8, *Mandatory* receives an input parameter because it is not evaluated at design time. *Max Length*, however, is evaluated at design time because, without it, the platform cannot display the widget with an incomplete visual representation. So, *Max Length* is not an expression, meaning it cannot accept parameters which only provide information at runtime. Thus, developers cannot customize properties outside a web block's definition that does not accept expressions, compromising the user experience.

In the example presented in Figure 3.7, the developer can pass one boolean parameter to make an Input widget inside the scope of the block mandatory or a number parameter for the width.

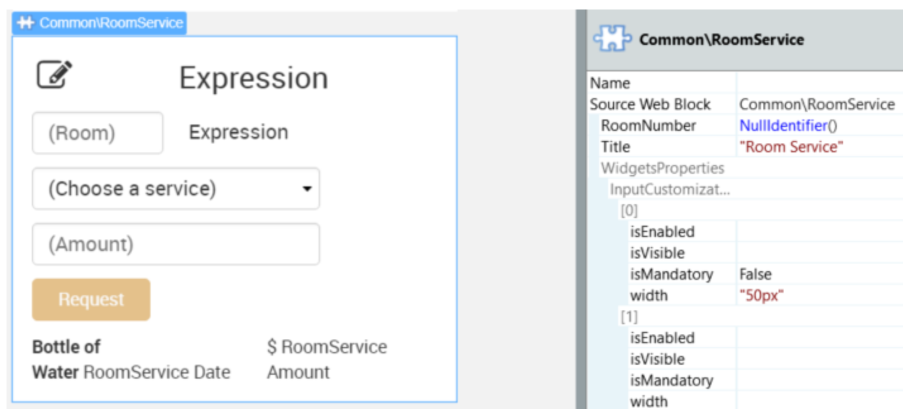


Figure 3.7: Web block instance with custom parameters.

To prepare web blocks for parametricity, parameters must be declared inside the block and then they have to be assigned to the widget properties inside the scope of it (Figure 3.8). For the example above, the boolean parameter will have to be assigned to the *Mandatory* parameter inside the Input widget. In the case of any style related parameters, the developer must declare what is called an *Extended Property*, which allows to customize certain properties with JavaScript expressions (can receive parameters). So,

for the width parameter, the extended style property must be declared in order to receive the parameter.

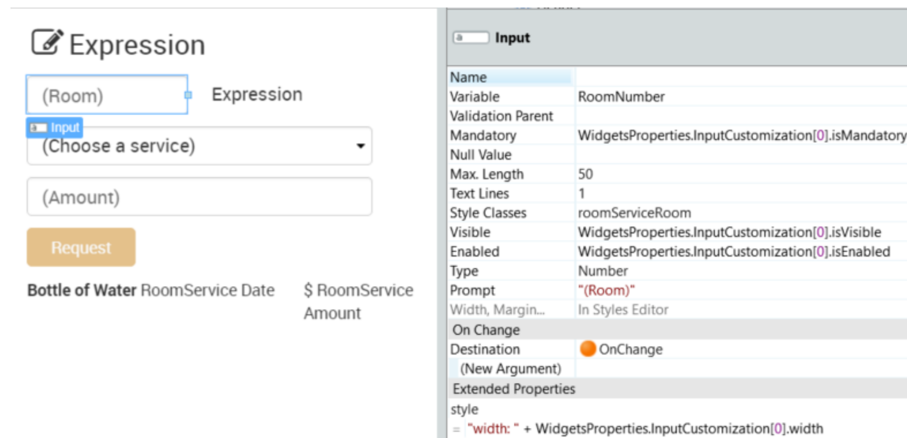


Figure 3.8: Input widget inside the Web block definition with parameters bound to it.

One of the advantages to this approach is that it makes the block fully customizable: it only requires the developer to change a parameter to customize one aspect of the block, but this ends up being a big weakness because if the developer wants to allow the customization of any widget of the block, then it will have as many parameters as the total of elements' properties. A very complex block with more than one Input, having every possible parameter declared, would make the properties editor overpopulated and very unappealing to edit.

The biggest disadvantage is that there is no preview on the WYSIWYG editor for the changes made by altering the input parameters; if the developer wants to preview their work they have to publish and only then check their changes by running the application.

3.3.2 CSS Styling

Another way to customize a web block is by the use of style sheets. However, this approach is applicable only to appearance-related properties.

The biggest advantage is one that is inherited by CSS: instant preview of the UI. The biggest disadvantage is also one that is inherited by CSS: CSS is hard to maintain. This approach requires the developer to know how to use style sheets and also maintain CSS selectors, that have to be unique. This is bad because people make mistakes. If developers have to maintain the code themselves, the platform cannot guarantee a consistent experience and prevent errors from happening.

To do this, the developer must create a unique selector and pass it to the *Style Classes* property of the element (*Style Classes* in Figure 3.8). Then, declare the style of the selector on the screen where the block is instantiated (Figure 3.9).

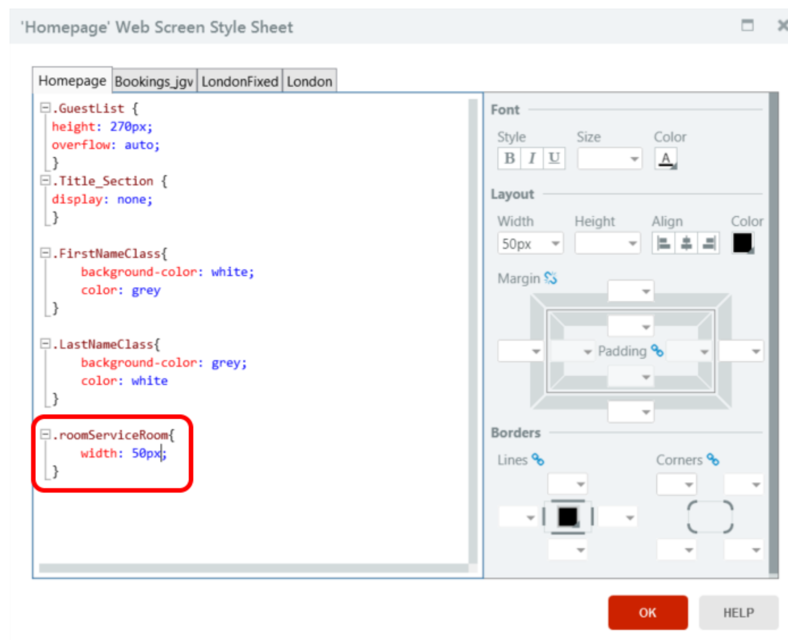


Figure 3.9: Screen's style sheet with customized selector (red box).

3.3.3 Duplication

It's easy to follow temptation and just duplicate the web block and customize it for another application, but it's not advised, as it is a way to deal with things rather neglectfully.

The advantages are quick customization of the block inside it's scope and its usage on another web screen (it can be the same used previously). The preview works (that's why it is tempting), but only because customization is done where it still was possible (web block's definition).

But the disadvantages can outweigh the advantages very quickly: if the developer were to create every possible scenario for the web block it wouldn't be possible, because they would have to create a duplicate for every one. Also, there would be a lot of code repetition. This is bad because if the developer wants to change the implementation of one of the duplicates, they have to change in every one of them.

3.3.4 Comparison between Workarounds

One approach can be evaluated from 7 stand points: only requires one block, is consistent with the platform, most reusable (allows to customize and updates every instance), has preview, no extra knowledge required, can change logic and can change style.

Duplicate can be used for faster implementation, but it will cost developer's time when changing one of the duplicates, because then they will have to change them all. Input parameters are good for reusability, but take a lot of time to make a feature customizable; they are best for logic features and not for styling as it doesn't allow preview. Style sheets allow to customize blocks very quickly and with preview, so, for this end, it is the best

Table 3.1: Approaches comparison. Best ones are the ones that have the most check marks.

Approach	Input Parameters	CSS Styling	Duplication
Only one block	✓	✓	-
Consistency	✓	-	-
Most reusable	✓	-	-
Has preview	-	✓	✓
No extra knowledge	✓	-	✓
Logic	✓	-	✓
Style	-	✓	✓

option; only works for styling though, so no logic parameters can be changed in this way.

Even though the workarounds presented enable some customization in web blocks, the disadvantages compromise all. None gives the developer the user experience that the rest of the platform provides. However, each has their piece of the puzzle, which, combined, would provide the ideal solution we are looking for. Input parameters expose the web blocks ability to pass information to the widgets inside; CSS styling provides a preview consistent with the platform; and Duplication shows how much quicker it is to customize a screen (customization of a web block's definition offers the same user experience as a screen's definition) than a web block.

Chapter 4

Related work

OOne of our objectives is to standardize a visual language so that it is not only easier to use, but also achieves the highest levels of quality. Support for variability in user interface development is important because it allows creating products that are more specific to the user needs.

Customizing user interfaces helps in this respect to create more valuable products. Since the programmer manually enters widgets customization, composition customization should follow the same rule so that all use cases are still possible. For this, we used the parametrization paradigm to pass the new values to the properties of the widgets to be customized.

The work related to this solution we aim to introduce can be divided into two categories: composition models and Visual Programming Languages (VPLs) for front-end development.

4.1 Composition Models

Java Beans

Java is an Object-Oriented Programming Language and became popular because of the concept of applets: small applications that were launched from a web page, motivating reusability of small parts and composition to extend a web application. However, composition models had some issues (e.g., lack of static type checking of applets being called from inside other applets) [20].

JavaBeans are software components (beans) that were an improvement over applets in terms of composition because they were made to be reused in many applications. One extra feature allows users to construct beans through visual tools [23]. Beans have the following aspects [20]:

- **Event** Beans can listen or trigger events (Connection 2.2.2).
- **Properties** Exposed instance properties with getter and setter methods responsible for changing the bean's behavior that can trigger events and also be constrained.
- **Introspection** Properties, events and methods can be inspected from the outside by an assembly tool (e.g., Jasmin [5])
- **Customization** "Exposed properties could be modified at design time by a property editor or bean customizers"[23]
- **Persistence** Customized bean instances are serializable: can be saved and restored

In Listing 4.1, we present an example of a bean implementation, where its state (instance variable *theValue*) will be "saved by the JavaBeans persistence mechanism" [23]. For this to happen, every bean has to implement the *Serializable* interface, methods must be public and have to be *thread-safe*[1]. Events can be fired by beans to notify interested beans on something. When an event is fired by a bean, it has to be able to notify every listening bean.

```
1  public class MyBean implements java.io.Serializable {
2     protected int theValue;
3     public MyBean (){}
4     public void setMyValue( int newValue ) {
5        theValue = newValue;
6     }
7     public int getMyValue() {
8           return theValue;
9     }
10 }
```

Listing 4.1: Bean example [23].

JavaBeans can be white-box components, just like the ones we aim to model in this work. Exposed properties, methods and events are necessary in a white-box reuse because compositions need this information so that users can customize the inside components. Any Java program can be a JavaBean, but one of the differences with JavaBeans is that must provide information at design time to "edit its properties and customize its behavior" [23]. Similarly, web blocks will have to expose this information in order for the design tool (Service Studio) can interact with the block in real time.

4.1.1 *UI* Families

Similarly, Pleuss et al. [14] introduce a new concept, Family *UI*, which consists of full screens to meet user requirements, which tend to use similar screens in their applications. However, users can request unforeseen use cases by implementing the user interface. In

[1]"(...) prevents more than one thread from calling a method at any given time"[23].

this case, the user must enter the customization manually. For this, Pleuss et al. use templates associated with the user interface to customize it. Although the motivation is different – screen customization rather than widgets sets – the concepts are the same as presented in this paper: new templates are added where it can save the customization the developer wants to make to the object, be it a screen or a block.

4.1.2 Customization by input

Using parameterization as a basis to solve the problem presented here is not new. Lizcano et al. [7] introduced a composition model for user interface components following a parameterization paradigm. Component views are black-box, so future modifications can only be made through parameters. Therefore, it is advisable to create components already prepared for eventual customizations.

4.2 Composition in Visual Languages

Composition in itself means gathering parts and connect them in a certain way. We present a very known tool in the Graphical Design community that has a simple composition model for composing images with several parts (Photoshop), and a similar LCDP to OutSystems which tries to turn compositions into white-boxes (Mendix).

Photoshop

Visual languages are not exclusive to application development. Actually, first ones were created to develop graphics. *Photoshop* [1] is a raster (dot matrix data structure) graphics editor with an interactive experience where users can edit images. One big feature is the concept of layers that compose an image. Layers are blocking, which means if one is on top of the layer tree, than it is rendered instead of the other one below. So, despite renderization being in 2D, layers give another dimension for designers to compose their projects.

A big advantage in using layers is a user can separate and combine several images to create another one without loosing the ability to edit each one. The layer panel houses every layer of the project and follows a directory structure so that users can organize layers in folders. To select a layer, a user can either select it directly in the layer panel or in the canvas, but only if the layer is visible, and then the program will search for the first layer in that selected spot.

Smart objects are layers that preserve the image's source content allowing a nondestructive editing by the user and can be compositions of multiple layers. The composition model allows editing of the layers inside a smart object and after saving the changes Photoshop updates every instance [22]. Because of this, smart objects maintain a master definition and ensure reusability.

This is what is known as black-box reuse, because users cannot edit the inside layers of a smart object instance. However, this approach is closer to what we are trying to achieve than the one presented in the next Section, as this one preserves a main definition of the smart object.

Mendix

Visual front-end development solutions are increasing every year, and *OutSystems* is not alone. *Mendix* [8] is a software company that, much like *OutSystems*, provides an LCDP for Rapid Application Development. Their platform (*Mendix Modeler*) delivers a very different experience from *Service Studio*; project explorer resembles the old-fashioned File Explorer from Windows and has every resource of the architecture located in one place instead of separating each development layer components like *Service Studio*.

(a) Title is changed in definition

(b) Instance does not receive the update

Figure 4.1: Differences between a building block's definition and corresponding instance

Building blocks allow users to create their own reusable components from scratch, or can choose from existing templates of some popular use cases that after instantiation can change every aspect of the block. In Figure 4.1a, a user changes the title inside the building block definition, but the change does not get to the already existing instance (Figure 4.1b). This means the instance is instead a duplicate of the original and not linked to a central definition. Thus, when using one building block on a screen, the block is duplicated and used as a template, because changes to the original do not propagate to already existing instances.

This is not the composition model we will try to achieve, because when a building block is used and the connection to the original is lost, reusability is not guaranteed. The idea behind the use of web blocks in OutSystems was always to allow users to reuse screen parts that were maintained by someone else; if the connection between the used block and the block's definition doesn't exist, then consequent updates to the block are lost. The ideal model would be a mix of the customization offered by Mendix's copy paste paradigm, and the reusability of Photoshop's smart objects, which Photoshop keeps a main definition of the component and when changed, instances get updated.

Chapter 5

New Composition model

A model defines each component and composition. These models determine the rules in which components and compositions must follow so that they can be combined.

Changing component properties through a visual programming language carries out a chain of events before assigning the new values to the component. Likewise, changing composition properties requires a similar process. For the customization of inner components inside a composition to happen, and this chain to succeed, the underlying models must be first adapted to this new feature.

In this chapter, the current composition model in the OutSystems' platform is presented, following an introduction to the component model, which defines widgets, web blocks, and web block instances. Next, we present the new composition model and the changes made to the original one to allow customization on compositions. We also deliver a proof of concept that validates our approach.

5.1 Current model

In this section, we explain the current models in use and what is currently missing for direct customization through the canvas of reusable compositions to be supported. Before introducing the new composition model, it is necessary to explain the existing model for the description of user interfaces. The study of the composition model already in use allowed the better understanding of the problem in question, necessary to assert what was the best way to intervene accordingly.

Service Studio follows a similar model to the one in Figure 5.1 for its building blocks. The ContentNode is the top most component in the model and composes other building blocks into one. These can be either Screens or Web blocks.

The model for ContentNodes follows the OutSystems language, which is used to

change the model. In Figure 5.1, a very simple metamodel of the original is illustrated. Service Studio delivers many widgets, but here we highlight the Input and WebBlockInstance. Contrary to widgets, the web block is not instantiated directly into the canvas when it is dragged into another ContentNode. Instead, it is referenced by another class, WebBlockInstance, that follows a different model from the web block. Because of this, WebBlockInstance is considered a widget in Service Studio. The web block, when created, is unique and can be referenced by any WebBlockInstance, just like screens.

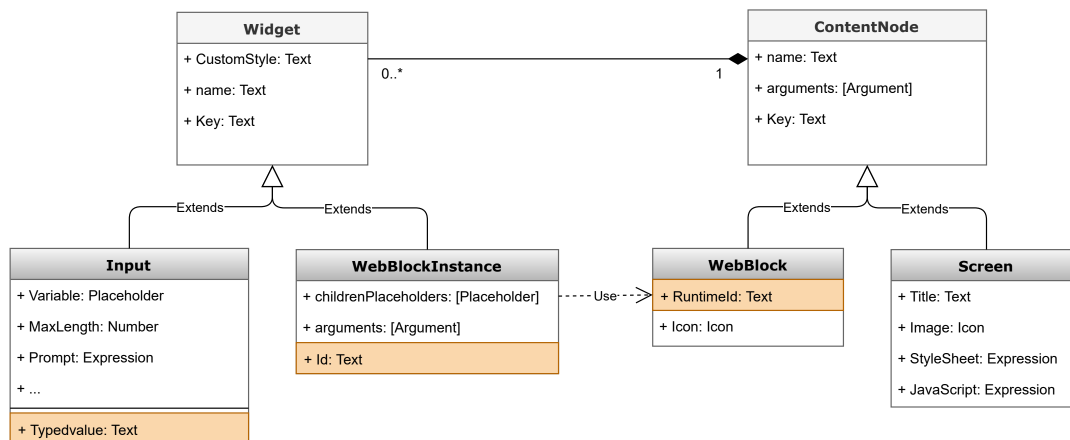


Figure 5.1: Model of OutSystems' building blocks.

In this simplified model, highlighted properties are run-time properties and the ones that are not are design-time properties. While their children widgets define web blocks, these inner components can not receive new values from outside the web blocks.

5.1.1 Widgets

Widgets follow a component model which dictates their description and how they can be composed either on web blocks or screens. Through properties, the developer can alter widgets as they best intend. Their simplified syntax in Figure 5.1 shows this, as the developer can pass parameters to determine the widgets' properties.

These properties can be design-time or run-time. What distinguishes one from the other is at what which point in time they are evaluated. Static properties are design-time because when they are assigned, they affect the preview instantly. Properties that can only be evaluated when the application is running or that are dynamic are defined at run-time.

```
<Input Key="1" Name="Input.ZipCode">
   <Design-Time Properties>
      <Variable Value="Address.ZipCode"/>
      <MaxLength Value=5/>
      <Prompt Value=Expression("55555")/>
      <CustomStyle Value=""/>
   </Design-Time Properties>
   <Run-Time Properties>
      <TypedValue Value=""/>
   </Run-Time Properties>
</Input>
```

Listing 5.1: Simplified widget model instanced for the "Input.ZipCode"widget in XML.

```
<input type='{{:Type}}' class='{{internalClass}}'
   placeholder='{{:Prompt}}' maxlength='{{:MaxLength}}' style='{{internalStyle}}'
   {{&internalProperties}} value='{{:Variable}}'>
</input>
```

Listing 5.2: Input widget HTML template.

For example, the widget for receiving information from the user into a text box - Input, Listing 5.1 - has several properties that are design-time with the exception of TypedValue that is run-time, which is where the inputted text from the user is to be saved, because its value is dynamic and can not be known at the point of design. Every widget class has a template for it to be displayed on to the canvas and a different one to be compiled into the end application. The template for the canvas of the Input widget is demonstrated in Listing 5.2. When the canvas instantiates a widget, the placeholders on the template are replaced with the respective properties and it is given to the canvas to be displayed.

5.1.2 Web Blocks

Web blocks are reusable compositions – the developer can define templates with multiple widgets without losing the reusability of a widget. Thus, web blocks are important to maintain a central definition so that changes made to these are seen on all of their instances. By following a standpoint in reusability, ensuring reuse of constructs guarantees good practices, resulting in faster development and less maintainability.

In this context, web block is the model to define reusable compositions. The WebBlockInstance class, which has a different model, defines WebBlock instantiation.

Widgets use existing templates to determine their appearances. Web blocks use the widgets templates for their appearance because widgets define web blocks. Therefore, the web block model is dependent on the widgets that define the web block and the widgets' visual representations that give web blocks their appearance. In Listing 5.3, a simplified web block model illustrates what defines web blocks. The widgets that compose them are the children.

```
<WebBlock Key="2" RunTimeId=123 Name="Address">
   <Icon Value="StreeSign">
   <arguments/>
   <children>
      <Widgets>
         <Input Key="1" Name="Input.ZipCode">
            <Design-Time Properties>
               <Variable Value="Address.ZipCode"/>
               <MaxLength Value=5/>
               <Prompt Value=Expression("55555")/>
               <CustomStyle Value=""/>
            </Design-Time Properties>
            <Run-Time Properties>
               <TypedValue Value=""/>
            </Run-Time Properties>
         </Input>
      </Widgets>
   </children>
</WebBlock>
```

Listing 5.3: Simplified web block model instanced for the web block "Address"in XML.

Although web blocks assure some reusability of patterns found in multiple use cases, these are insufficient to address all of the developers' needs, because just like screens, web blocks are static and do not generally allow to change their inner components from the outside - exceptions explained in section 3.3.

5.1.3 Web Block Instances

In Listing 5.4, a simplified model of web block instances is illustrated. This model references the web block which it will instantiate – the source web block. The visual representation of the instance is the same as the source web block's visual representation.

```
<WebBlockInstance Id=124 Key="3">
   <SourceWebBlock Key="2" Name="Address"/>
   <arguments/>
   <children/>
</WebBlockInstance>
```

Listing 5.4: Simplified web block instance model for the instance of the web block "Address"in XML.

This model is used when a user drags a web block onto a screen. Web blocks can not be instantiated since it is impossible to reference duplicate widgets. If a developer instantiated two web blocks on the same screen, then Service Studio could not reference their widgets because there would be two of every widget (Duplicate Key Exception). WebBlockInstance solves this by separating the two web blocks into two different instances. Although there are widget duplicates, Service Studio can not access them.

5.2 Enabling customization

A quick analysis of the model of reusable compositions shows that there is not a way to change their inner components. Therefore, the problem is finding a way to link the inner components to the outside scope.

A composition's representation is dependent on the uniqueness of its components because it references them at the component level. Thus, components need to be unique for conflict reasons - cannot reference a duplicated component. A new model to define how to instantiate web blocks is needed, because widgets in web blocks are unique to the web block's definition – if a web block were to be instantiated twice at the same scope, accessing the inner widgets would be impossible because they would be duplicated.

Since web block instances only reference the source web block, it is not possible to have instances with different visual representations of the same web block - different appearance. Thus, a new model for web block instances is necessary because among all instances of a web block, there is not a way to assign new values to each of the web block's children. The new model addresses this by incorporating new structures for each web block instance, in order to keep track of possible customizations made by the developer, as well as new structures in web blocks to assert what can be customized when instantiated.

5.2.1 Requirements

However, first, we determined the requirements for the customization that both user and creator of web blocks would be able to do and not do so that we could break apart the problem into smaller sub-problems and validate the prototype's success by checking the correctness of every solution to each sub-problem.

Anyone can define web blocks. Creators are the ones who define web blocks to be later used by themselves or another person. Web block users are the developers who use web blocks defined either by them or another person.

At this time, creators can only define black-box web blocks where users can only drag and drop these to their canvases without being able to change them directly. The only control the creator has on its web blocks is assigning input parameters to widget properties so that the user can input new values for these. On our approach, creators must be able to determine which widgets are customizable - can be changed - to control the ones they do not want to be changed. Visually, the customizable widget with customizable properties should respectively show what properties are customizable and the ones that are not. The ones that are not customizable should not appear or appear as they could not be customized - greyed out.

As for the user, right now, their ability to change web blocks is lacking. Ideally, users must be able to access and change customizable widget properties inside a web block's instance and get an instant preview of the changes, similarly to developing inside the web block definition. Also, widgets must be able to affect other widgets through dependencies

for a more intuitive experience (e.g., input widget below another one may reflect inherent the width of the upper one).

5.2.2 New Model

Concerning our requirements, new structures must be added to the existing models to save the new information needed to convey customization. In reality, the user will not be changing the original web block but instead changing a new representation of it.

As the creator cannot determine what can be customizable on their web blocks, a structure to establish what properties can be customized when instantiated - *CustomizableWidget*, Figure 5.2 - was added in the class that defines web blocks. A collection of these allows storing one *CustomizableWidget* per widget in the web block. This new class determines what widgets can be customizable. In here, another structure saves the customizable properties (*CustomizableWidgetProperty*) - only the name is necessary - and the parent widget, for search purposes. When using the web block, the properties present in the respective *CustomizableWidget* allow the user to change their duplicate representation on the web block instance.

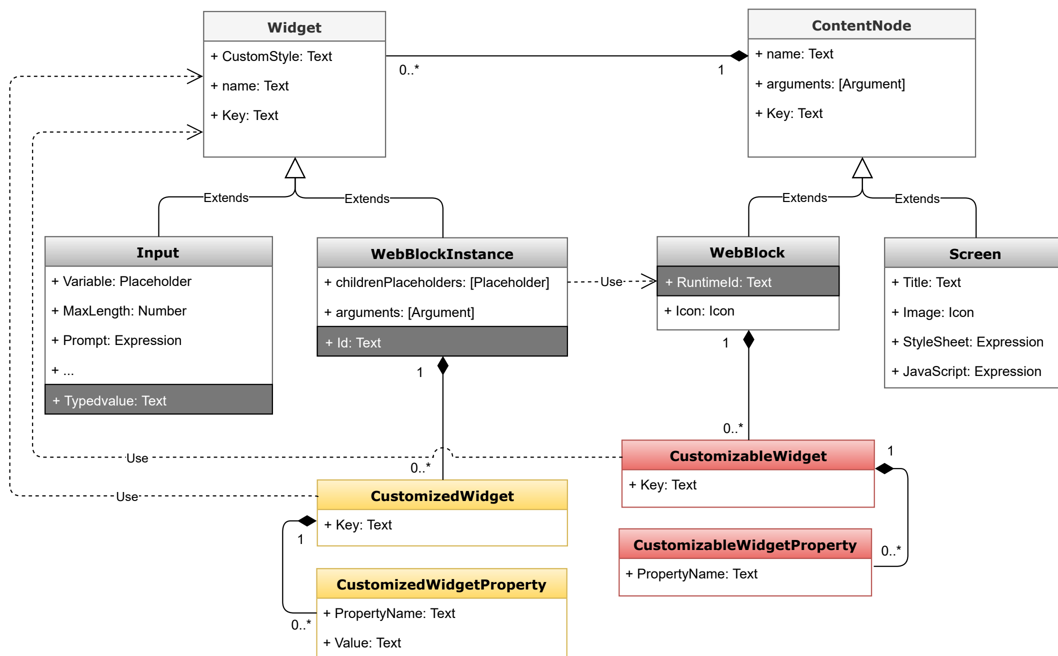


Figure 5.2: New structures added to support web block customization.

The web block instance has properties distinct from other instances of the same *web block*, so it has its class to represent it. However, the definition retains the composition's visual representation, and it is necessary to save the values for customization in the instance.

Similarly to the WebBlock class, in the WebBlockInstance class, the structure *CustomizedWidget* (Figure 5.2) was added to save the new customizations that the user desires

to make to a web block. The new structure saves every user customized widget, which is represented with another class, *CustomizedWidget*, allowing the connection between the widget to customize and the respective properties. The difference to the *CustomizableWidgetProperty*, is that the *CustomizedWidgetProperty* links the property name to the new value inserted by the user to customize the property.

In our solution, the new value placeholder is a simple string variable, but the best way to make the model fully integrated would be to make the new value an expression. Expressions are compositions of operands and operators that need to be evaluated beforehand to get a value. This allows attributing calculations directly to properties without the need of assigning to a static variable. By turning the new value placeholder into an expression, the user could assign it any calculation, as long as it had the same type of property to customize.

5.3 Model comparison

To better understand how our approach affects the current model in use, we illustrate the differences between before and after the adaptation with a use case. A possible use case relevant to this problem could be the following: in a home delivery application developed on Service Studio, it is necessary to request the user's address. It makes sense that the form in Figure 5.3 is a web block – Address –, as it is used on more than one screen. However, initially, the web block was designed for the U.S. address, which has a five-character zip code, while the Portuguese address has eight.

Figure 5.3: Web block for requesting a user's address - Address web block.

In the current model, the Address web block has the order of components presented in Listing 5.5. The widget Input that receives the zip code of the user is on line sixteen, and it has a maximum allowed length of five – the widget Input can not receive more than five characters.

```
<WebBlock Name="Address">
   <Form Name="Form1">
      <Container Name="Input_Name">...
      </Container>
      <Container Name="Input_Street">...
      </Container>
      <Container Name="Input_ZipCode">
         <ChildWidgets Count="2">
            <Label>
               <Text Value="Name"/>
            </Label>
            <Input Name="ZipCode" MaxLength="5"/>
         </ChildWidgets>
      </Container>
      <Container Name="SaveButton">...
      </Container>
   </Form>
</WebBlock>
```

Listing 5.5: Address Web Block's child widgets in the current model.

```
<WebBlock Name="Address">
   <CustomizableWidgets>
      ...
      <CustomizableWidget Key="xxxxxxx">
         <Widget Name="ZipCode">
         <CustomizableWidgetProperties>
            <CustomizableWidgetProperty>
               <PropertyName>"MaxLength"</PropertyName>
            </CustomizableWidgetProperty>
            ...
         </CustomizableWidgetProperties>
      </CustomizableWidget>
      ...
   </CustomizableWidgets>
   <Form Name="Form1">
      <Container Name="Input_Name">...
      </Container>
      <Container Name="Input_Street">...
      </Container>
      <Container Name="Input_ZipCode">
         <ChildWidgets Count="2">
            <Label>
               <Text Value="Name"/>
            </Label>
            <Input Name="ZipCode" MaxLength="5"/>
         </ChildWidgets>
      </Container>
      <Container Name="SaveButton">...
      </Container>
   </Form>
</WebBlock>
```

Listing 5.6: Address Web Block's child widgets in the new model.

If we want to customize the widget for the input of the user's zip code, it is not possible without having to change the original model and affecting all web block instances, which we do not want.

In the new model, Listing 5.6, the collection *CustomizableWidgets* allows the creator to define what can be customized in the web block. We need this because otherwise, the web block user could customize anything. Before the web block creator had to actively define input parameters and assign them to widgets so that the user could customize them. That control is essential for the web block experience, and with this, we can enable just that.

The current model for web block instances is defined in Listing 5.7. The instance is a child of the Homepage screen, and references the Address web block. From here, it is also impossible to customize the web block, since the widget WebBlockInstance only references the web block and passes input parameters.

```
1  <Screen Name="Homepage" Key="5" Title="Homepage">
2     <Image Value="">
3     <Stylesheet/>
4     <JavaScript/>
5     <WebBlockInstance Key="xxxxxxx" SourceWebBlock="Address">
6        <Arguments>
7           ...
8        </Arguments>
9        <Placeholders>
10          ...
11       </Placeholders>
12       ...
13    </WebBlockInstance>
14 </Screen>
```

Listing 5.7: Address Web Block instance in the current model.

With our approach, we can successfully store the new values that will replace the default ones of the widgets the user wishes to customize. On line eleven of Listing 5.8, the CustomizedWidgetProperty for MaxLength is defined with the value "8", which on a later time, will replace the default value "5", stored in the Input widget ZipCode of the original web block, in both the view and compiled application.

```
<Screen Name="Homepage" Key="5" Title="Homepage">
   <Image Value="">
   <Stylesheet/>
   <JavaScript/>
   <WebBlockInstance Key="xxxxxxx" SourceWebBlock="Address">
      <CustomizedWidgets>
         ...
         <CustomizedWidget>
            <Widget Name="ZipCode"/>
            <CustomizedWidgetProperties>
               <CustomizedWidgetProperty>
                  <PropertyName>"MaxLength"</PropertyName>
                  <Value>"8"</Value>
               </CustomizedWidgetProperty>
               ...
            </CustomizedWidgetProperties>
         </CustomizedWidget>
         ...
      </CustomizedWidgets>
      <Arguments>
         ...
      </Arguments>
      <Placeholders>
         ...
      </Placeholders>
      ...
   </WebBlockInstance>
</Screen>
```

Listing 5.8: Address Web Block instance in the new model.

5.4 Summary

We presented the existing model and its compromises facing customization of compositions. The lack of support for new values to be assigned to widgets inside web blocks required changes to the model.

The new model takes this into account, with the introduction of new structures capable of containing the new information relative to possible customizations done to web blocks by the user. Not only to save customizations in separate constructs, but to allow a safe environment where web block creators have the power to dictate what can be changed on their reusable compositions and maybe more important what can not.

Our approach would be complete if we used Expressions instead of Text for the new values that the new structures can handle. But it would require more resources and the aim of this thesis was to validate that it was possible to customize reusable compositions. It was more important for us to deliver a solution that scaled through the entire development of an application then mastering every part.

The proof of concept in Chapter A helped us prove our solution was feasible, allowing us to proceed adapting the OutSystems platform. Furthermore, using the *React* library proved application model could be compiled to the end application, since the platform compiles directly to *React*.

Chapter 6

Implementation

In this chapter we discuss the implementation of the prototype into Service Studio. At first, it was important to achieve a solution that expanded through the development of an application to prove its feasibility.

For this, we made a list of tasks to complete, splitting the development of an application in Service Studio into several phases. In order to implement direct customization to web block instances with an instant preview, interactive experience, and also support to compile the application model to the end application ready to be deployed, these phases were organized as follows:

- Architecture: Service Studio's architecture follows an MVC pattern - Model-View-Presenter -, and changing all parts was necessary to reach our goals;
  - Model: changing the underlining models to support firstly from the lowest level of implementation that disables customization;
  - View: extend platform's View to support the changes previously made;
  - Presenter: new ways to request changes in the Model and to receive requests from the View;
- Compiler: compile the application model into a deployable application ready to receive customizations and to pass new values for web block instances.

Having a prototype that expanded not only to the development phase of an application but also to the end phase of compiling allowed the further development into the interaction aspect of the platform, which at that point was done by a couple of commands.

The prototype still needs some work to fully support every use case expected, which will be presented in the end.

6.1 Architecture

Although this thesis' expectations are to deliver a new composition model capable of customizable reusable compositions, is also very important to deliver a solution capable of handling every aspect of visual programming languages. Not only the Model was adapted, but also the three layers of the architecture used in Service Studio, since changes were needed in the canvas the developer sees, the way they can interact with it, how the platform processes the developer's request and how the platform responds.

Service Studio's architecture follows an *MVP* pattern (Model-View-Presenter) as a way for the developer to create applications (Figure 6.2). The more known *MVC* (Model-View-Controller) is a similar design pattern used for implementing graphical user interface objects (Figure 6.1).

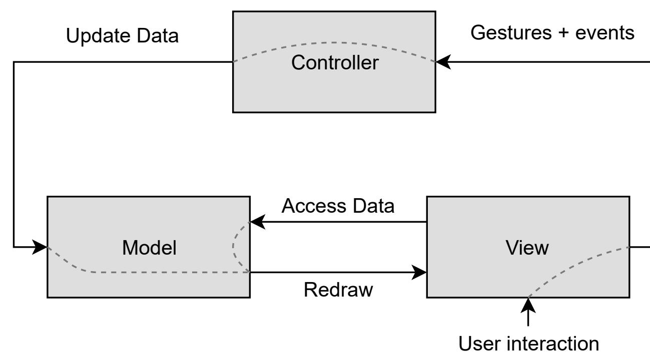


Figure 6.1: *MVC* design pattern.

In both patterns, three different parts split a common programming problem: implementing a user interactive system. The user interface objects have associated data that helps drawing them on the screen. This data is stored in the component Model and then the View uses the data to draw the object on the screen. In the *MVC* pattern, the controller changes the object's data stored in the Model as the user interacts with the View with gestures and by triggering events [15].

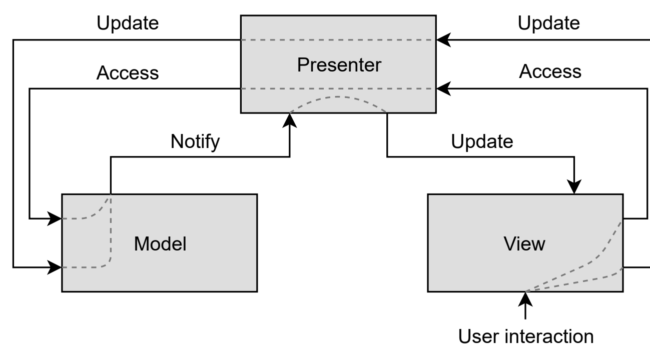


Figure 6.2: *MVP* design pattern.

In the *MVP* pattern, the controller is replaced with the Presenter, becoming a middleman between the Model and View - both only communicate with the Presenter and never with each other. Because of this, *MVP* has several benefits for separating data management from user interface (e.g., encapsulation allows for more maintainability between the Model and View).

In Figure 6.3, it is shown what happens when a developer tries to change a widget, where the developer's action propagates through the platform and returns back to them. When the developer changes an object (e.g., widget) by interacting with the *View*, an event is raised. These events are in turn subscribed by the *Presenter* which uses *Commands* to modify the *Model*. The *Presenter* uses generation counters for every object to keep track of changes, so when an object's data is changed in the Model, the respective generation counter will be incremented, resulting in a refresh of the *View*.

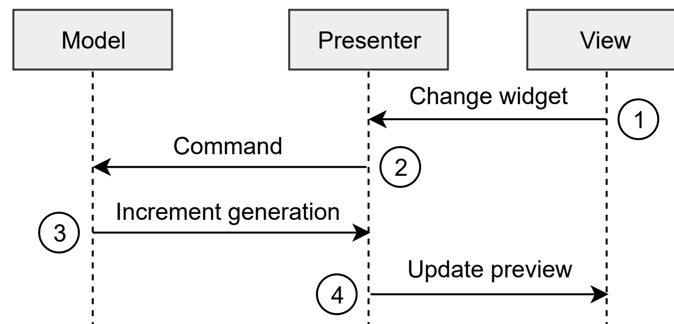


Figure 6.3: Time line of when a widget is changed.

Because of encapsulation, adapting the Service Studio to our Model was easier than if it would have been using an *MVC* paradigm. It was not necessary to change how the Presenter interacted with the Model or the View with the Presenter, but only how each processed the requests made by each other.

6.1.1 Model

After designing the new Model, we proceeded to adapt the current composition model to support directly customization of web blocks. The existing architecture allowed an easy integration with the new designed structures to save the new information related to customization. The platform keeps a record of many types of building blocks in a separate file in order to generate the code for each one, requiring only the platform developer to change this file - ObjectDefinitions.

```
<!-- CustomizableWidget object definition -->
<AbstractObject name="CustomizableWidget">
  <Properties>
    <Property name="Widget" type="AbstractWidget"/>
  </Properties>
  <Children>
    <Child type="CustomizableWidgetProperty" prop="CustomizableWidgetProperties"/>
  </Children>
</AbstractObject>
<AbstractObject name="CustomizableWidgetProperty">
  <Properties>
    <Property name="PropertyName" type="Text"/>
  </Properties>
</AbstractObject>
<!-- CustomizedWidget object definition -->
<AbstractObject name="CustomizedWidget">
  <Properties>
    <Property name="Widget" type="AbstractWidget"/>
  </Properties>
  <Children>
    <Child type="CustomizedWidgetProperty" prop="CustomizedWidgetProperties"/>
  </Children>
</AbstractObject>
<AbstractObject name="CustomizedWidgetProperty">
  <Properties>
    <Property name="PropertyName" type="Text"/>
    <Property name="Value" type="Text" affectsPreview="true"/>
  </Properties>
</AbstractObject>
```

Listing 6.1: New structures added to ObjectDefinitions.

In Listing 6.1, the new classes to represent widgets on web blocks or web block instances, *CustomizableWidget* and *CustomizedWidget*, are added to ObjectDefinitions in the form of *XML*. In the case of *CustomizableWidget* - determines if a widget is customizable inside a web block's definition - the property *Widget* stores the widget that is referenced. Besides this, the *CustomizableWidgetProperty* is also referenced with the name *CustomizableWidgetProperties*, as this is a collection stored inside the customizable widget. In this context, the term *child* means the object can store multiple objects of the same type into a collection, which is referenced through a property. This collection references the class *CustomizableWidgetProperty*, which only stores the name of a widget's property that can be customizable.

For the case of *CustomizedWidget* - determines the new values to be given to a widget in a web block's instance - it is also store the reference to the widget the web block instantiates, and the collection of *CustomizedWidgetProperties*. This collection references the class *CustomizedWidgetProperty*, which similarly to the *CustomizableWidgetProperty* class, storing the name of the property to be customized, it also stores the new value to replace the property's default one.

The tag affectsPreview determines if the property should increment the object's generation counter when it is changed. If the affectsPreview is true and the property is

changed, then the generation counter for that property's object will be incremented and the preview will eventually be refreshed. The tag is used in the property *Value* because the property can be changed several times inside the *CustomizedWidgetProperty*. For this, it was necessary to extend the compiled class of *CustomizedWidgetProperty* to store a self preview generation counter that would be incremented if the property were to change. When the property changes, a request to refresh the preview is propagated to the property's parent, and then to the parent's parent until it reaches the root, and only then will Service Studio check if the preview is to be refreshed - in this case, it would be.

```
<Node name="WebBlock">
   <Properties>
      <PropertyPlaceholder name="Public"/>

      <!-- Group: Events -->
      <ChildPlaceholder name="OnInitialize"/>
      <ChildPlaceholder name="OnReady"/>
      <ChildPlaceholder name="OnRender"/>
      <ChildPlaceholder name="OnParametersChanged"/>
      <ChildPlaceholder name="OnDestroy"/>

      <!-- Group: Advanced -->
      <ChildPlaceholder name="StyleSheet" />
      <ChildPlaceholder name="ScriptDependencies" />
      <ChildPlaceholder name="JavaScript"/>
   </Properties>
   <Children>
      <Child type="JavaScript" property="JavaScript" />
      <Child type="CustomizableWidget" property="CustomizableWidgets"/>
   </Children>
</Node>
```

Listing 6.2: New web block class definition.

It matters only to properties that will change overtime because properties that do not change can not influence the preview. *PropertyName* is static - it is the same from the beginning till the end of the *CustomizableWidgetProperty* and *CustomizedWidgetProperty* classes -, which means that the preview is only influenced by the creation of the parent objects.

Defining which widgets are customizable and customizing them requires changes in the preview and adjacent editors of Service Studio. Therefore, instantiating either one of the classes requires incrementing the respective generation counter.

In the Service Studio architecture there are two types of compositions that can be instantiated in an application model - web blocks and screens. These compositions follow a tree paradigm inside the Model, where each one can be viewed as a node.

In Listing 6.2, the class for web blocks is adapted to fit our requirements to reach our new composition model. The new child represents the new collection CustomizableWidgets and references objects of type *CustomizableWidget*. When a web block creator determines a widget and respective properties to be customizable, a *CustomizableWidget* object

is created and stored inside the *CustomizableWidgets* collection, as well as a *CustomizableWidgetProperty* object is created for every property that is to be customizable.

```
<AbstractWidget name="WebBlockInstance">
   <Properties>
      <Property name="SourceWebBlock"/>
      <ChildPlaceholder name="Arguments"/>
   </Properties>
   <Children>
      <Child type="Argument" property="Arguments"/>
      <Child property="PlaceholderArguments"/>
      <Child type="CustomizedWidget" property="CustomizedWidgets"/>
   </Children>
</AbstractWidget>
```

Listing 6.3: New web block instance class definition.

The *WebBlockInstance* class in Listing 6.3 references the web block in the SourceWebBlock property, since the instantiation of web blocks must be through another class. The new child CustomizedWidgets will allow to store new values for the web block widgets. When a web block user customizes a web block instance, a *CustomizedWidget* object is created and stored inside the *CustomizedWidgets* collection, as well as a *CustomizedWidgetProperty* object is created for the new value given by the user to be stored in the object.

6.1.2 View

The View is the part of the *MVP* paradigm which the user interacts with. Through this component, the user gets a visual representation of the Model - preview - and can change it by interacting with it. Currently, the View writes the preview in HTML, and the View transforms each object in the Model to a visual representation in HTML to later insert into the preview. However, we want to check first for customizations on widgets inside web block instances so that we can replace the preview of the original widget with a new one.

After adapting the Model so that it could retain customizations of web blocks, we could use the new Model to affect the View. We can change the View so that it can show a preview of the customized widgets because we have access to children objects of web blocks and web block instances.

Service Studio's View component uses templates to render each widget, and each type of widget has its unique template. The templates are written in HTML and have placeholders for the properties that the developer can change inside the platform - placeholders are between the double curly brackets. When rendering a widget, the View will access the template for that type of widget (e.g., the Input widget in Figure 5.2) and replace its placeholders with the widget's properties.

After the Model changes, the Presenter is notified, and in turn, it notifies the View to refresh. The way that the View is refreshed is through a visitor pattern, where it begins with the highest node of the application model and does a depth-first search [2] of every

widget. When the View arrives at a widget, it requests the widget's template and formats it by replacing the placeholders with the widget's data respectively.

Therefore, to show a widget inside a web block instance customized as the user wants, the formatted template must have the new values before it is rendered. To achieve this, it is necessary to replace at the time the template is being formatted or after, provided the default value are replaced before returning the final format to render. Before formatting would mean changing the widget of the original web block, which we do not want. We want to preserve the modification of web blocks only to their creators.

Algorithm 1
Checks for pending customizations and replaces the properties' default values.

1: $widgetRepresentation \leftarrow currentState.WidgetRepresentation;$
2: $customizedProperties \leftarrow newDictionary < string, string > ();$
3: $sourceWidget \leftarrow widgetRepresentation.SourceWidget;$
4: **if** $sourceWidget$ not equal $null$ and $visitedWBI.Count > 0$ **then**
5: $\quad cedWidget \leftarrow visitedWBI.Peek()).CustomizedWidgets.Where(w => \{$
6: $\quad\quad w.Widget.Equals(sourceWidget).FirstOrDefault();$
7: $\quad\quad$ **if** $cedWidget$ not equal $null$ **then**
8: $\quad\quad\quad$ **for each** $c \in cedWidget.CustomizedWidgetProperties$ **do**
9: $\quad\quad\quad\quad customizedProperties.Add(c.PropertyName, c.Value);$
$\quad \});$
10: $htmlElement \leftarrow FormatHtmlTemplate(widget, customizedProperties);$

The Algorithm 1 is a simplified version of what was implemented to achieve this. The currentState relates to the state of the visitor. *WidgetRepresentation* (source widget) is the widget it is on at any moment. Because the source widget can be null, we must check this - if the visitor is at a node, the source widget will be null.

Next, we check if the widget belongs to a web block. *VisitedWBI* stores the last web block instances visited. If there are any, the first one is the last one, which means the widget belongs to the source web block of that web block instance.

By accessing the last visited web block instance, it is possible to access its *CustomizedWidgets*. So, the next step is checking for a CustomizedWidget on *CustomizedWidgets* with its widget equal to the source widget.

If there is one, then we search for customized widget properties and add them to the customizedProperties collection. Lastly, we send the collection along with the widget that holds the template to be formatted (FormatHtmlTemplate), so that when the placeholders are replaced with the respective properties in the collection, the new values replace the placeholders rather than the default ones.

6.1.3 Presenter

In an *MVP* paradigm, the Presenter is the middle layer between the Model and View. Developers interacting with the screen triggers events which are sent from the View to

the Presenter. Every event sent to the View is sent to the Presenter, be either a change or an access to the Model, because the Presenter subscribes these events. The Presenter then issues requests to change the Model and after receiving confirmation from the Model it sends an update to the View.

The next step of the solution development was allowing the developer to interact with the widgets inside a web block instance. In Subsection 5.2.1, we discuss the requirements needed to be achieved. The web block creator should be able to determine somehow that a widget can be customized.

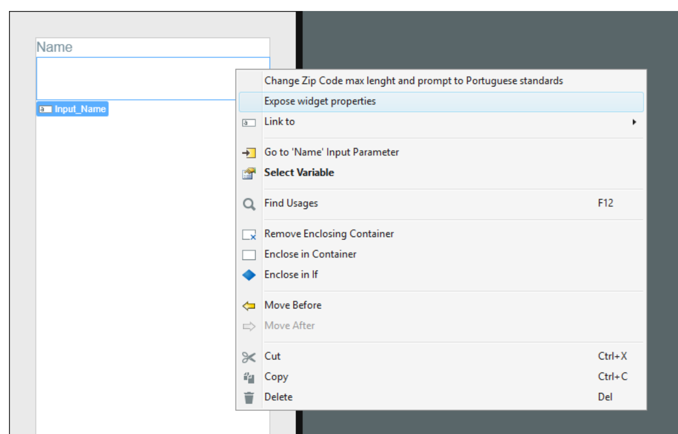


Figure 6.4: The command turns the widget *Input_Name* properties customizable.

To request the Model to change, the Presenter uses Commands. The Algorithm 2 is the operation the Presenter requests when the user chooses from the dropdown menu in Figure 6.4, when selecting a widget inside a web block definition. This way, only the web block creators can determine what is customizable.

It is necessary to pass a widget as target, since the command is bound to a widget This widget will be the starting object in which the command will execute its procedure. The widget's parent must not be *null*, because the command will only execute on widgets that belong to web blocks.

In this case, the command will request the Model to insert the widget selected in the new created collection *CustomizableWidgets* inside the parent web block. Because of this, the command traverses through the widget parents until it finds a web block.

When the parent web block is found, a new *CustomizableWidget* pointing to the initially selected widget is assigned to the web block so that the new *CustomizableWidget-Property*s can be assigned to the former. Considering it is created a *CustomizableWidget-Property* for every widget property, every property belonging to the widget will become customizable.

Algorithm 2
Command to expose a widget's properties to the outside of a web block.

```
 1: procedure Command(Widget target)
 2:     if target.Parent not equal null then
 3:         parentWebBlock ← widget.GetParent();
 4:         while parentWebBlock is not WebBlock do
 5:             parentWebBlock ← parentWebBlock.GetParent();
 6:             if parentWebBlock equal null then
 7:                 return null;
 8:         cw ← newCustomizableWidget(parentWebBlock);
 9:         cw.Widget ← widget;
10:         for each prop ∈ widget.CustomProperties do
11:             p ← newCustomizableWidgetProperty(cw);
12:             p.PropertyName ← prop.PropertyName;
13:     return null;
```

Now that web blocks are customizable, developers that will use these do not still have a way to interact with web block instances. As stated in Subsection 5.2.1, the user should be able to access customizable properties of customizable widgets inside a web block instance.

The next point of focus was the selection of widgets inside web block instances. Selecting widgets and the whole interaction supported in Service Studio does not take into account web blocks. A user can only select the web block instance on a screen, where as for other widgets on the screen, the user can select them, change their properties and drag and drop widgets around the screen. To solve this, the Presenter had to be changed in order to support the selection of every widget, since it is responsible for processing what happens after an event is triggered by the View.

When something is selected, Service Studio can get the exact or closest object to where the user selected, triggering an event which is subscribed by the Presenter. The Presenter then receives a representation of the widget - *WidgetRepresentation* - which it will insert into the selected objects collection. The reason why a collection is used is that it is possible to select more than one object. The class *WidgetRepresentation* apart from the widget, it also stores the path to the root of the widget. This is necessary because the widget only stores its direct parent, which in the case of web blocks, the widget would reference the parent web block and not the web block instance. It is essential to know the web block instance where the widget is being instantiated because when selecting the widget, it is fundamental to know if the selected widget was selected inside the web block definition or in one of its instances. When the object belongs to a web block, it is the web block that is inserted into this collection. Because of this, it is impossible to select a widget, since the web block is selected instead.

By inserting the *WidgetRepresentation* of the selected widget and not its web block parent into the selected objects collection, the Presenter will show the correct selected

widget when the checking for selected objects. However, we do not want to select all widgets, only the customizable ones. The Algorithm 3 returns a selected widget if it is customizable or not for it to be edited in the properties editor (Figure 6.5). To check this, we have to traverse the *WidgetRepresentation* parents until we reach a web block instance. If not, the procedure returns null, and the widget can not be customized or selected. If the procedure gets to a web block instance, the *CustomizableWidgets* collection is checked for having the selected widget. By being inside the collection, the widget is customizable and can be selected.

Algorithm 3
Returns selected widget if it is customizable.

```
1:  selectedWidgetRepresentations ← view.SelectedWidgetRepresentations;
2:  return selectedWidgetRepresentations.Select((widgetRepresentation) => {
3:      widget ← widgetRepresentation.Widget;
4:      parent = widgetRepresentation.Parent;
5:      while parent not equal null and parent.Widget is not WebBlockInstance do
6:          parent ← parent.Parent;
7:      if parent not equal null then
8:          webBlockInstance ← parent.Widget;
9:          webBlock ← webBlockInstance.SourceWebBlock;
10:         customizableWidget ← webBlock.CustomizableWidgets.Where(w =>
11:             w.Widget equal widget);
12:         if customizableWidget not equal null then
13:             return widgetRepresentation.Widget;
14:         else
15:             return null;
16:     return widgetRepresentation.Widget;
17: });
```

After selecting the widget, the developer should have access to the object's data and permission to customize it. The properties editor in Figure 6.5 is responsible for showing widget properties and was adapted to the new Model since we want to access the newly created structure - *CustomizedWidget* -, to check for new values to replace the original default ones assigned to the widget.

Properties have structures to define them, and widgets have several properties on their behalf. When a widget is selected, the properties editor accesses the widget's properties and displays them. However, selecting a widget that we do not want to change, but rather change a representation of it - *CustomizedWidget* -, we must first select the representation, and only then we can change its data.

The Algorithm 4 is responsible for checking if there are customizations from the user and creating new properties for these to be given to the properties editor. Traversing the widget parents until reaching a web block instance ensures that the widget is from a web block and thus is not meant to be changed. When a web block instance is reached, the

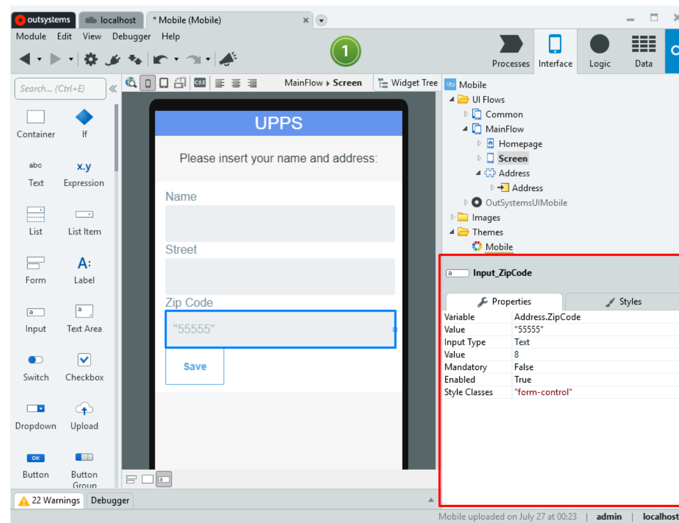


Figure 6.5: Properties Editor with *Input_ZipCode* selected.

next step is to check that the widget is customizable by accessing the *CustomizableWidgets* collection inside the *SourceWebBlock* referenced by the web block instance. If the widget is not customizable, then the properties editor will be empty.

Following this, we check for customizations - searching for the customized widget of the selected widget inside the web block instance. If a *CustomizedWidget* does not exist for the selected widget, then the properties editor will be empty. If not, each widget property will be given a new property with the propDescriptor of the respective property of the selected widget, and the customized widget object as its parent.

Every property has a propDescriptor which has the entire description of the property. Giving this to the new property ensures nothing from the original property is lost, and we only have to assign the new value to the new property. The alternative would be to copy all data values one by one from the original to the new property. New properties for every widget property is necessary because when the developer changes them, the request to change them must be sent with the new property and not the original.

Now that we can change a widget inside a web block instance without changing the original widget inside the web block, the View must be updated so that it can show the developer in real-time the changes they make through the properties editor.

To update the View, objects must have generation counters, that they can increment after being changed, to request for a refresh. In Subsection 6.1.1, affectsPreview determines that if the property is changed, then the preview should be updated. The newly created properties do not have this feature because it is exclusive to higher hierarchy objects - *CustomizedWidgetProperty*.

To solve this, the properties' parent must have a generation counter that increments when the property changes. Thus, *CustomizedWidget* class received a generation counter to increment when one of its properties changes.

Algorithm 4
Checks for pending customizations and replaces the properties' default values.

```
1:  widget ← widgetRepresentation.Widget;
2:  parent ← widgetRepresentation.Parent;
3:  while parent not equal null and (parent.Widget is not WebBlockInstance) do
4:      parent ← parent.Parent;
5:  wbi ← parent.Widget;
6:  wb ← webBlockInstance.SourceWebBlock;
7:  if parent not equal null and wb.CustomizableWidgets.Where(w =>
8:      w.Widget equal widget) is caw then
9:      customizableProperties ← caw.CWPs[a].Select(cawProp =>
10:         cawProp.PropertyName);
11:     cProperties ← cProperties.Where(prop =>
12:         customizableProperties.Contains(prop.Name));
13:     if wbi.CustomizedWidgets.Where(ceW =>
14:         ceW.Widget equal widget) is CustomizedWidgetcw then
15:         cProperties ← cProperties.Select(prop => {
16:             propDescriptor ← prop.PropertyDescriptor;
17:             newProp ← newProperty(propDescriptor, cw);
18:             if cw.CustomizedWidgetProperties.Where(
19:                 cwProp => cwProp.PropertyName equal prop.Name) is cwp then
20:                 newProp.Value ← cwp.Value;
21:             return newProp;
            });
```

[a] CustomizableWidgetProperties.

6.2 Compiler

The last step of application development is deployment. Developers can do this by just clicking a button in Service Studio, but behind it all, much work is being done. From the beginning till the point of deployment Service Studio stores a simplified model of the application so that it can be sent to the Platform Server and be the basis of a new application built upon current technologies that are ready for every browser or smart device. This process contains compiling which, by accessing the Model of the application developed in Service Studio, formats templates of every building block with the data obtained from the Model into a fully deployable application.

Just like what was done in Subsection 6.1.2, where the View formats web block templates with the data stored in the web block instances - Figure 6.6 -, we want to compile the application model into an application that supports web block customization. However, the View replaces the default values on widget properties with customizations actively, which means for the same screen with two web block instances of the same web

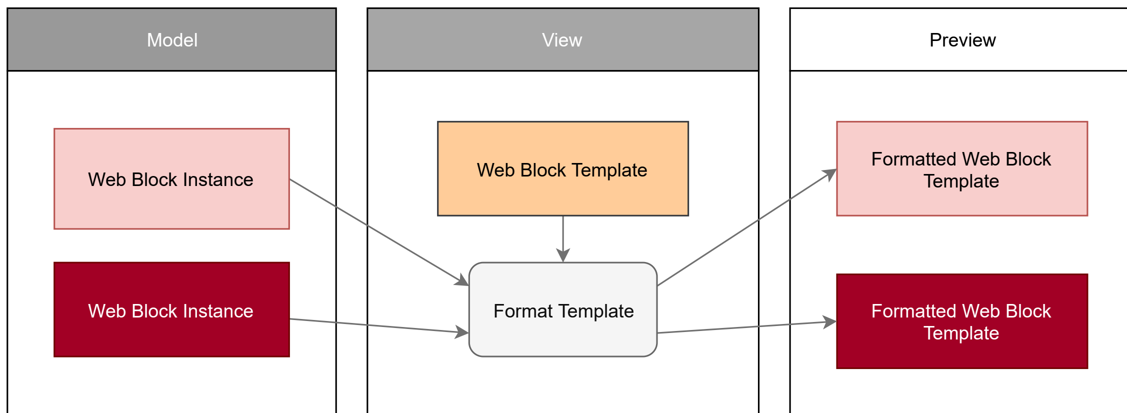


Figure 6.6: How the preview is drawn by the View.

block, the generated preview of both web block instances are different. The Compiler can not do this because web blocks are compiled into components that are instantiated onto screens, which means it is impossible to have two different web block instances of the same web block - Figure 6.7.

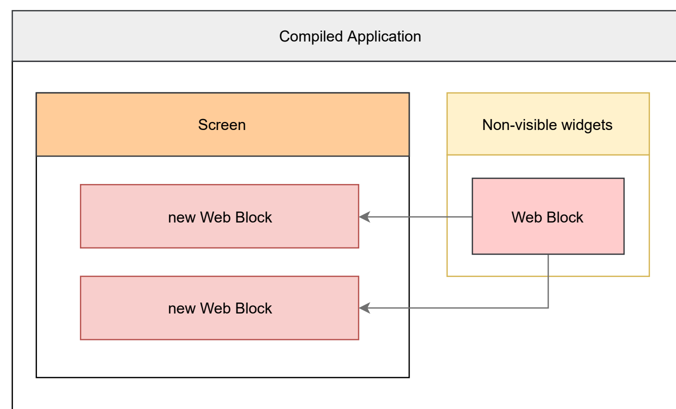


Figure 6.7: How the compiled application draws its web blocks.

The compiled artifacts are written mostly in JavaScript, and all building blocks remain as whole units because the Compiler uses the React library to structure the application compositionally. In this case, web blocks are compiled into React components, which screens or other web blocks will instantiate - no web block instance class needed. Therefore, we needed a way to customize in real-time instances of web blocks because we did not want to create more than one component for a web block: that would not be feasible as an application can have many instances of one web block.

To solve this, we parameterize the web block components, and store the customizations where the web block instance would be in the original application model (e.g., screens). Then, screens or web blocks can pass the new values to the web block components where they can assign these to the respective widgets. So, it was necessary to determine a way to assign these new parameters to the inner components of web blocks,

and how screens or web blocks store these.

Current Compiler

Since web block components can be instantiated many times, and used in many applications, assigning parameters to the respective widgets is essential as to guarantee all possible customizations. However, if there are widgets that are not customizable, we do not have to assign any parameters to customize them, saving resources and complexity. Ideally, we would want to only assign the parameters that we know a screen or web block will use to pass customizations. Widgets can be customizable but may not have been customized, so not creating parameters and not having to assign them would decrease resources used. However, complexity may have an impact, since we had to look up for every customized widget to determine what parameters could be ruled out. We did not choose to follow this path, and instead assign all customizable widgets with parameters, to keep our solution simple.

Following this, we had to change the Compiler to check for customizable widgets and generate the components accordingly, so that at run-time web block components are ready for any customization coming from their instances, and their customizable widgets assigned to the respective parameters or default values. If the widget is not customized, we want to assign the default value.

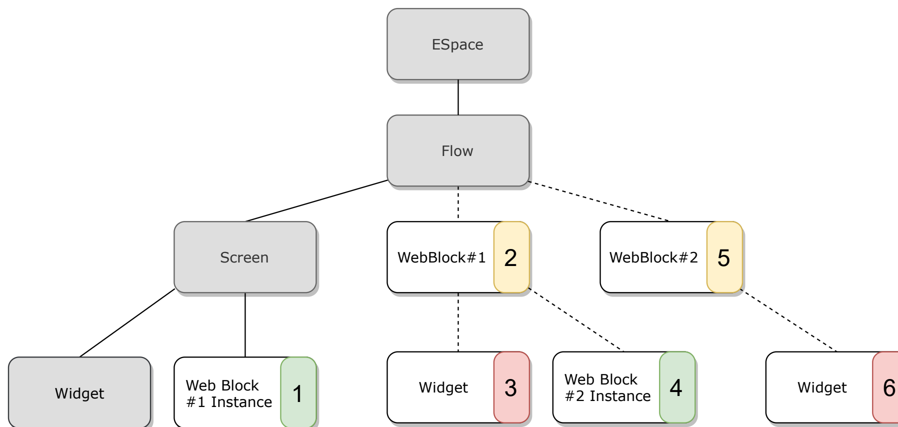


Figure 6.8: Order of compilation for the components of an application model.
: CompileBlockInstance Straight line : branch traversed
: CompileBlock Dashed line : branch not yet traversed
: CompileWidget

The Compiler follows a visitor design pattern where it compiles each widget as its traversing the application model tree, just like the View in Subsection 6.1.2. However, as the View keeps a record of the nodes above the one it is visiting, the Compiler does not. Although it is possible to access the highest parent node of a widget, by traversing through its parents, in the case of widgets belonging to web blocks, this only gets us the parent web blocks, and not the web block instances that store the customizations. So

when a widget from a web block is being compiled, we can not check directly if the widget is customizable, let alone access the customizations stored in the web block instance.

Figure 6.8 demonstrates how the Compiler traverses an application model tree. *ESpaces* are the root of an application model, and they store everything related to an application needed for the Platform Server to compile into a deployable application. *Flows* are groups of compositions - web blocks and screens -, so that the developer can organize their applications by user interfaces and logic that is more closer together [21]. After traversing the tree, the Compiler reached the web block instance, which is the first step of the demonstration.

The Compiler starts by compiling the web block instance into the artifact shown in Listing 6.4, which is a call to instantiate the Address web block. In here, for simplicity, only a few parameters are sent to create the element, but no parameter to customize the web block. Elements in React are immutable objects that describe component instances [18]. These only have two fields:

$$type : (string \mid ReactClass) \text{ and } props : Object.$$

So, the first parameter is the type, which is the React component for the Address web block, and the second are the properties that describe the component.

The web block is not compiled just yet since the Compiler will visit screen to screen and web block to web block. When compiling widgets, the Compiler has reached the end of that tree path and will move on to other nodes. Unless it is a web block instance, which can have placeholders, in that case, the Compiler will traverse to those.

```
1 View.prototype.internalRender = function () {
2
3     return React.DOM.div(React.createElement(AddressWebBlock, {
4       id: { uuid: "1" }
5     }));
6 };
```

Listing 6.4: Screen in Figure 5.3 from the *UPPS* app compiled.

AddressWebBlock is the compiled component in Listing 6.5 and has all the code needed to create elements for every widget inside the web block. The first element is to create the element for the component Form, a compiled widget provided by another script, as all other widgets are. Inside, a container is created with a label and an input inside. The input is responsible for the input of the zip code by the user. All the properties that belong to the widget in the application model are written in the properties of the new element.

```
View.prototype.internalRender = function () {

   return React.DOM.div(this.getRootNodeProperties(),
      React.createElement(OSWidgets.Form, {
         _idProps: { name: "Form1" }
         }, (...)
            React.createElement(OSWidgets.Container, {
               id: { uuid: "7" },
               React.createElement(OSWidgets.Label, {
                  targetWidget: "Input_ZipCode",
                  id: { uuid: "8" }
               }, "Zip Code"),
               React.createElement(OSWidgets.Input, {
                  enabled: true,
                  inputType: /*Text*/ 0,
                  mandatory: false,
                  maxLength: 5,
                  prompt: "55555",
                  id: { name: "Input_ZipCode" }
               })
            }), (...)
   }));
};
```

Listing 6.5: Web block *Address* from the *UPPS* app compiled.

New Compiler

Understanding how the current Compiler works allowed us to develop a solution that works, using what has been already done and adding new logic to ensure customization control is guaranteed and safe, but most importantly that changes made by the user can be deployable. So that developers can customize web blocks, publish their applications, and see their results on the end application. The three already mentioned procedures in Table 6.8, are used to compile the respective building blocks. Each one was modified to enable customization.

Since the Compiler does not store a state of the tree while traversing it, we need a state to store at least the last node visited. When visiting a widget, we will only need the web block, or node, directly above to check if it is customizable. Thus, in Figure 6.8, in step one, the last node visited would be the screen, in step three, the first web block, and in step six, the second web block. In our new model, web blocks determine what is customizable in their scope, which means a web block does not keep a record of what is customizable on another web block, allowing us to only need the last web block visited.

To access customizations stored in web block instances, we also need to keep a state for the last visited web block instance. Thus, in step three, the last web block instance visited would be the first web block instance, and in step six, the last one would be the second web block instance. Just like web blocks, their instances do not store information of other objects just because the objects are in their scope, in this case, customizations of other instances, since they only store the customizations for the web block they instantiate. So,

we only need the last web block instance visited to customize a widget of that web block.

The Algorithm 5 portraits the new procedure for compiling blocks, whereas of now it keeps a state of the current content node being visited. After saving the web block, the View is compiled. In this case, View means screens or web blocks. The boolean *isBlock* determines if the View is a screen or a web block. *CompileView* is also used to compile screens, so the first parameter, isBlock, determines if the node is a web block or a screen.

Algorithm 5
New procedure for compiling web blocks.

1: **procedure** CompileBlock(contentNode)
2: $currentCN \leftarrow contentNode;$
3: $CompileView(/*isBlock*/true, contentNode);$
4: $currentCN = null;$

Inside the *CompileView* procedure, the Compiler visits every child node. Only when it visits a widget that it calls *CompileWidget* and calls the procedure in the Algorithm 7. When the widget is a web block instance, the Compiler calls the procedure *CompileBlock-Instance* in the Algorithm 6. In here is where we store the web block instance to have access to potential customizations later when visiting the widgets inside the web block. Just like *CompileView*, *CompileBlockInstance* sends a flag, isWebBlockInstance, to indicate to the *CompileWidget* procedure that it is a web block instance.

Algorithm 6
New procedure for compiling web block instances.

1: **procedure** CompileBlockInstance(widget, wbi)
2: $currentWebBlockInstance \leftarrow wbi;$
3: ...
4: $CompileWidget(widget, /*isWebBlockInstance*/true);$
5: ...-
6: $currentWebBlockInstance = null;$

Before, the Compiler transformed widgets into components only by their properties, without accessing data from other building blocks. Now, we want to access their parent web block, if they have one, and if so, the web block instance where they have been instantiated. In Algorithm 7, we demonstrate the changes made to the *CompileWidget* procedure, now with *currentCN* and *currentWebBlockInstance*, guaranteeing access to the last visited node and web block instance.

CompileProperties on line four, is the function which transforms the widget's properties into the wanted standard:

$$< PropertyName, Value >.$$

This way, we can reference the property by its name or key. The function receives the new empty dictionary *props* in order to fill with the new data type properties, and receives the

widget properties.

Next, we need to check if the widget is a web block instance or an atomic widget(i.e., a widget with a visual representation and can not compose any other widgets). If it is an atomic widget, then we search for potential customization on the last visited web block instance so that we can compile the widget based on the new values given by the user.

At the same time, it is crucial to check what kind of node is stored in the state, since the last visited node can be a screen and not a web block. CurrentCN can also be null if the widget is inside another data type other than a screen or a web block. Ideally, we check if it is not null, and if it is, we know the widget does not have any potential customizations since it is not inside a web block. Then, we check if it is a web block and assign it to a new variable if so, *wb* on line five. Entering the *if* statement means the procedure is in the presence of a widget.

Now that we have the web block parent, we access to its *CustomizableWidgets* collection to check if the widget is customizable. If the collection contains the widget, then we proceed to check the *CustomizableWidgetProperties* collection of the found customizable widget object for existing customizations. However, it is necessary to discard properties that have no value expression associated. For simplicity sake, we do not customize these properties.

For every customizable widget property found, the procedure formats it into the following string:

```
string.Format("this.props.customizedWidgets &&
   this.props.customizedWidgets[\"{0}\"]\"  {1}, widgetProperty, defaultValue)
```

Listing 6.6: The new formatted property value.

The new string allows us to check the new structure, *customizedWidgets*, for any customized widgets when the application is running – if we called right away from the structure for a specific property and it did not exist, it would trigger a null pointer exception. The dictionary *customizedWidgets* is created while the procedure is compiling a web block instance. After compilation, the property would be presented as follows:

```
maxLength: this.props.customizedWidgets && this.props.customizedWidgets["
     Input_ZipCode.MaxLength"]  5
```

Listing 6.7: The new compiled property.

The first placeholder with the tag "0"in Listing 6.6 replaces with the string "Input_ZipCode.MaxLength", which is the way we designed the *props* dictionary so that we can access the respective property with ease. The second placeholder with the tag "1"replaces with the default property value. If the widget is not customizable, *customizedWidgets* will be empty, and the expression will go for the default value, in this case, "5". If it is customizable, but the property was not customized, then the expression will be

false in the second parcel and will decide for the default value. In case that the customizable widget property does not exist, then the property will be compiled as it was already before.

Algorithm 7
Checks for pending customizations and replaces the properties' default values.

```
1:  procedure CompileWidget(widget, isWebBlockInstance)
2:      ...
3:      props ← new Dictionary < PropertyName, Value > ();
4:      props ← CompileProperties(props, widget.props);
5:      if widget is AtomicWidget aw and currentCN is not null
6:          and (currentCN is WebBlock wb) then
7:          cw ← wb.CustomizableWidgets.First(cwt => cwt.Widget equal aw);
8:          props ← props.Where(p => p.Value is not null).Select(p => {
9:              if cw is not null then
10:                 cwp ← cw.cwProps.First(cwpt => cwpt.pName equal p.Name);
11:                 if cwp is not null then
12:                     return {
13:                         "this.props.cws is not null and this.props.cws[\"
14:                         <cW.Widget.Name>.<cwp.pName>\"]" not null
15:                         ,p.Value.Value
                        }
16:                 if p.Value.Value is not null then
17:                     return {p.Name, p.Value.Value};
18:                 else
19:                     return null;
          });
20:     else if isWebBlockInstance then
21:         cProps ← newDictionary < PropertyName, Value > ();
22:         for each w ∈ currentWebBlockInstance.CustomizedWidgets do
23:             for each wp ∈ w.CustomizedWidgetProperties do
24:                 cProps.Add({
25:                     " < w.Widget.Name > . < wp.pName > ",
26:                     wp.Value
27:                 });
28:         props.Add("customizedWidgets", cProps);
29:     else
30:         props ← props.Where(p => p.Value is not null);
31:         props ← props.Select(p => {prop.Key, prop.Value});
32:     ...
```

If the widget is a web block instance, the procedure will enter in line 20 of the algorithm to create a new structure, *customizedWidgets*, to store the customizations of that web block instance. We search for all customized widgets inside the last visited web block instance until we get the correspondent customized widget, and then for each property, the procedure adds the property to the new structure. If the widget is not in a web block,

then the procedure enters the last *else* statement, where it cleans up the *props* dictionary for any properties without any expressions and transforms it into the correct standard.

```
View.prototype.internalRender = function () {
   return React.DOM.div( React.createElement(OSWidgets.Form, {
      _idProps: { name: "Form1" }
      },(...)
         React.createElement(OSWidgets.Container, {
            id: { uuid: "7" },
            React.createElement(OSWidgets.Label, {
               mandatory: false,
               targetWidget: "Input_ZipCode",
               id: { uuid: "8" }
            }, "Zip Code"),
            React.createElement(OSWidgets.Input, {
               enabled: true,
               inputType: /*Text*/ 0,
               mandatory: false,
               maxLength: this.props.customizedWidgets &&
                  this.props.customizedWidgets["Input_ZipCode.MaxLength"]  5,
               prompt: this.props.customizedWidgets &&
                  this.props.customizedWidgets["Input_ZipCode.Prompt"]  "55555",
               id: { name: "Input_ZipCode" }
            })
         }), (...)
   }));
};
```

Listing 6.8: Address component in Listing 6.5 after adaptation.

In Listing 6.8, the Address web block from Listing 6.5 is demonstrated with the new changes implemented. *MaxLenght* and *Prompt* are customizable now, and at run-time, if there are new values, they will be chosen instead of the default ones. The other properties do not have this access to the new structure, and therefore can not receive customizations.

```
View.prototype.internalRender = function () {
    return React.DOM.div(React.createElement(AddressWebBlock, {
      customizedWidgets: {
         "Input_TextVar.MaxLength": "8",
         "Input_TextVar.Prompt": "88888888"
      },
      id: { uuid: "1" }
   });
};
```

Listing 6.9: Screen component in Listing 6.4 after adaptation.

In Listing 6.9, the screen from Listing 6.4 now has a new structure to send to the web block. Through here can pass as many values as there are customizable widget properties. The syntax is simple: the property name comes first so that it is easy to find inside the dictionary, and the new value comes in second.

Chapter 7 Evaluation

After we finish implementing the prototype, we began by evaluating if our solution worked and contributed to the OutSystems Platform. It is essential to establish this to validate the work we have done until now.

The following sections will be explaining the methodology used to evaluate the success of the solution; then we will proceed to the analysis, and finally, we will present the results, as we will discuss them.

7.1 Usability Testing

Given that our problem is based on visual interaction, usability testing is very important to validate our work. Because of this, we gathered a group sample to represent all OutSystems developers, with the only important difference being their experience using Service Studio since not many people know how to customize web blocks.

In this work, it is important to not only score how our approach is intuitive but also to compare with what is currently offered by OutSystems. Since we want to compare the two approaches, we divided the group sample into two: the Beta group and the Alfa group. The Beta group used the current approach by OutSystems to customize web blocks, whereas the Alfa group used our approach.

After this, we created a script to guide each participant to do a simple task and achieve the same outcome. We present the script in Appendix B. The script is made of a description and a task at the end. Both groups were given the same description of an application where it was required to change a field inside a web block. The only conditions given to both groups were they could not duplicate the web block or change other screens that were instantiating the web block. Since we do not want to influence each group differently in any way, both tasks have to meet equal outcomes. However, we want participants in the

Alfa group to achieve the task by using our approach. Alfa group could achieve the task by choosing the current approach delivered by OutSystems, so we added as a condition that they could not access the web block definition, since creating parameters in the web block requires accessing the web block definition.

After all, participants finished their task, we requested them to fill in a System Usability Scale quiz. The questions are presented in Appendix C. The SUS score helps us determine if our approach is intuitive and also helps us compare with OutSystems' current approach. The score is calculated by the following equation:

$$score = ((I-1)+(5-II)+(III-1)+(5-IV)+(V-1)+ (5-VI)+(VII-1)+(5-VIII)+(IX-1)+(5-X))*2,5 \quad (7.1)$$

Another important metric is how much time did the participant take to accomplish the task. The time was measured from the moment the participant understood the task to the moment they published the application and verified the task was complete.

In table 7.1, we show the respective SUS scores and times for both groups. In Figures 7.1 and 7.2 we plot these results, with the respective standard deviations. With our approach participants completed their tasks in less time, with a difference of 5.45 times on average. SUS scores also increased with our approach, with one of the participants scoring 100 and another scoring the lowest with 87.5, the average resulted in 95. As for the Beta group, the average SUS score is 37.5, the lowest being 17.5 and the highest 50. The scores were not as concentrated as the Alfa group, but they all fell under the fifties. The scores can be viewed in Appendix C.

	Average Beta	**Average Alfa**
SUS (0 – 100)	37,5	95
Time (s)	300	55

Table 7.1: Average SUS and time taken to complete the task for each group.

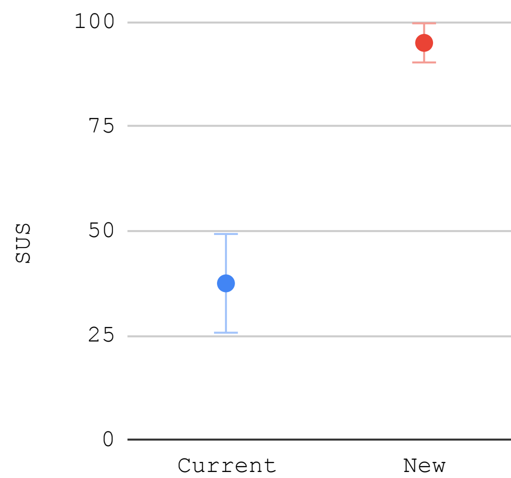


Figure 7.1: Average and standard deviation of SUS for each group (higher is better).

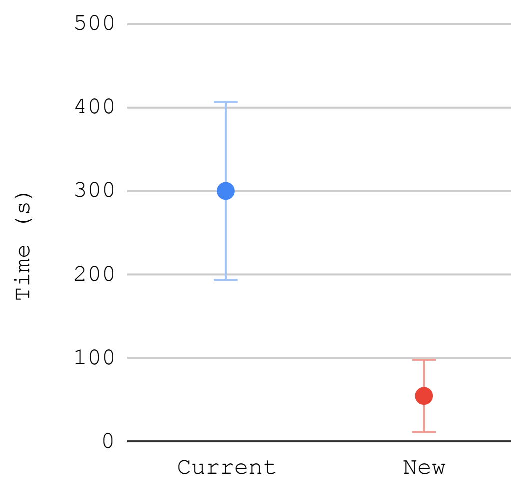


Figure 7.2: Average and standard deviation of time taken to complete each group's task (lower is better).

As expected, the Alfa group took less time to complete the task and scored our approach as more intuitive than what the Beta group scored the currently offered solution. Note that it was the first time the participants had to interact with our approach, so we expect experienced participants to need even less time to finish the task.

The Alfa group participants also commented widgets should be highlighted when they are selected, an aspect we did not implement. We verified some of the participants did not understand the widgets were being selected, and because of this took more time to finish the task. Because of this, we can expect even less time required to complete the task on a more fluid experience with widgets being highlighted when being selected.

7.2 Solution comparison

In this section, we compare a solution offered by the OutSystems' repository of reusable modules and user interface components, with our approach. We analyzed several widgets, which in reality are disguised web blocks, on how it allows customization on reusable compositions. The results can be observed on Appendix D.

To validate our approach, we began by analyzing what is offered currently by Service Studio, and then how much we can improve on customization. Besides giving us an insight on how customizable can these widgets be, it also allowed us to perceive the work necessary by the platform developers to continue to implement new widgets on an architecture that is outdated and with a much due revision.

Our study included 29 of the 64 widgets of the OutSystemsUIMobile module. The criteria to analyze was the following:

- Similar Widgets: many of the UI patterns have duplicated widgets with small differences;
- Property Parameters: input parameters used to link the user directly with widget properties;
- Action Parameters: input parameters used for actions;
- Style Parameters: input parameters used on stylesheets;
- Potential Customizable Widgets: widgets inside web blocks that could be customized but are not;
- Preview: does the input parameters affect the preview when changed.

7.2.1 Analysis methodology

We analyzed the widgets inside the module based on the criteria previously mentioned, to measure how much customization is allowed on them.

One of our objectives since the beginning was to reassure reusability was not affected. However, some widgets offered in the module have similar widgets, with small differences. These should account for customization, since duplicating a widget because there is not another way to allow customization, is, in fact, a compromise on customization.

Input parameters are used on many of the widgets provided, and these have varied on how they are used. We divided these into three groups: property parameters, action parameters, and style parameters.

Another one of our objectives was to make the preview reactive to the changes made to widgets inside web blocks. So, if the preview is affected or not is another factor to take into account when measuring customization.

7.2.2 OutSystemsUI

OutSystemsUI is a module that offers several pre-built widgets that in reality are compositions. But, they bring many limitations. Widgets reference pre-built web blocks. The user customizes these through pre-defined input parameters on each web block. When instantiating several widgets, we can analyze their design is not uniform – some refresh the preview, while others do not. None have styles associated because web blocks do not have them.

In comparison, our solution is intuitive – delivers what is expected. The creator should be able to create widgets just like the ones offered in OutSystems, and the user should be able to select whatever widget they want.

7.2.3 Examples

These are some of the widgets the module provides to the developer, each with their own compromise.

Adaptive UI Columns

Adaptive UI Columns are patterns for the developer to implement a multi-grid layout inside their application. These are pre-built web blocks of many column arrangements, ending up with widgets very similar. However, with our approach, we could create more columns, each with different sizes.

Badge/UserAvatar

The badge allows putting a number inside a round background; good for counters. When changing its properties, the preview is instant, which means it has been hard-coded into the platform.

The number and background color are the only things that the user can change. It would be nice to change its size. With our approach, we can change everything inside.

Icon Badge

This widget is composed of a placeholder with a badge on top of it. As observed in the widget Badge, we can change its background color and the number it shows through input parameters. However, the widget Icon Badge cannot link its two input parameters with the widget Badge because input parameters are only evaluated at run-time. The widget Badge can show instant preview when changing its parameters because its developer hard-coded it into Service Studio to refresh the preview.

7.2.4 Comparison

The gathered results from our study of 29 of the 64 widgets of the OutSystemsUIMobile module can be seen in the Appendix D. Service Studio offers many use cases in the form of pre-built static web blocks. However, these have limited customization. Our approach is ideal for customization, since comparing the both ours deliver the aspects offered in every pre-built widget.

Based on our evaluation, our approach is expected to help not only developers, but also platform developers, because our approach fills a hole inside Service Studio, and completes the whole process flow between module developers and users.

Input parameters will still be important because they allow the user to enable some feature in the widget by toggling it, instead of having to design the logic behind it. The user only has to ask the widget to do something without worrying how it is done.

Conclusions

This work presents a new composition model inspired by the present model of the OutSystems platform. The resulting prototype proves that it is possible to customize components within a composition, without compromising the preview of changes and without losing the uniqueness of the definition of a block.

The results we obtained prove our solution gives users more freedom to do what they want, but not an exclusive alternative to the platform. All mechanisms provided by the platform help create an environment easy to develop with, and with efficiency and quality in mind.

Impact analysis

Analyzing the results, we gather there is a significant gain for platforms like OutSystems by adopting our solution.

From the usability tests, our approach proves developers customizing reusable compositions can gain more than 80% of their time compared to using what is currently offered by OutSystems. Not only this, but the participants also scored our approach as being more intuitive and easy to use, with higher than double the score given to what is currently offered.

By analyzing the OutSystemsUI module, which was created by specialized developers, we conclude our approach can help with many of the implementations offered in this module while saving up resources like hiring specialized developers to create modules highly specialized and not that much customizable. However, some widgets in this module offer other possibilities that with our approach we can not achieve. Meaning many of the patterns implemented provide widgets with a complex logic that is difficult for a citizen developer to implement. This allows more customization since these web blocks

offer more use cases where before developers wanted but did not have the resources.

Lastly, with our approach, we can still have a live preview, whilst OutSystemsUI is not consistent throughout its widgets. For widgets to have a live preview, a specialized developer is necessary to hard-code to refresh the view when an input parameter changes. As shown in Appendix D, almost half of the widgets have an inconsistent live preview.

Future Work

As for future work, it is expected to build a more comprehensive interaction, meeting the needs of the creators and users of compositions. Also, customizations with run-time properties would allow new situations to explore, as discussed in Section 3.1.

Our approach would allow direct customization of web blocks, but at the moment, there are still several things to be worked on in order to make our solution reach the level we would like to achieve.

Model

Right now, the template is prepared for design-time properties, but not for run-time properties, because the customization is saved as text. As for the next step, we intend to use expressions to extend customization to all properties. This way, it would also be possible to introduce dependencies between widgets. This problem is interesting because passing information between components creates new opportunities: one component will inherit properties from another automatically (e.g., width); a programmer having access to one component's data model and being able to insert it into another (e.g., one component sharing one user's data access to another component would decrease redundant accesses).

Presenter

The Presenter is responsible for receiving requests by the View and process them accordingly. When web blocks creator wants to make their web blocks customizable on our solution, they only need to select the widget and select the command to make it customizable. In the future, we would like to have more control over what can be customizable. Not only the widget, but we would also like to decide if we only want a property to be customizable.

As for the users, at the moment, they can only customize some properties in the widgets. Ideally, they could customize all properties that were decided to be customizable by the creator and not be restricted by only those that are supported.

View

From the beginning, we knew interaction would be the final stage of development since we had to have a general solution that covered the entire process of developing applications.

Because of this, interaction is the part with the most work to be done. Still, what we offer right now enables us to reach inside web blocks and customize their inner widgets, something that was impossible to do from the beginning. This was difficult at first since the architecture is not built to handle the widgets inside the web blocks. However, we managed to get past some obstacles, and now it would be easier to implement what is left.

At the moment, interaction is mostly done by menus, especially from the creator's point of view. Selecting widgets still is not perfected. It only works in some use cases.

Compiler

The compiler was the most developed part of this work, besides the model. However, we would like to have the opportunity to test with expressions instead of using text. It would have allowed many possibilities because most properties use expressions.

Bibliography

[1] *Adobe Photoshop CC | Best photo, image, and design editing software*. URL: https://www.adobe.com/products/photoshop.html (visited on 02/17/2019).

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009. ISBN: 9780262033848.

[3] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds. *Structured Programming*. London, UK, UK: Academic Press Ltd., 1972. ISBN: 0-12-200550-3.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[6] K.-K. Lau and T. Rana. "A Taxonomy of Software Composition Mechanisms." In: *Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*. SEAA '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 102–110. ISBN: 978-0-7695-4170-9. DOI: 10.1109/SEAA.2010.36.

[7] D. Lizcano, F. Alonso, J. Soriano, and G. López. "A New End-user Composition Model to Empower Knowledge Workers to Develop Rich Internet Applications." In: *J. Web Eng.* 10.3 (Sept. 2011), pp. 197–233. ISSN: 1540-9589. URL: http://dl.acm.org/citation.cfm?id=2230836.2230838.

[8] *Low-code Application Development Platform - Build Apps Fast & Efficiently | Mendix*. URL: https://www.mendix.com/ (visited on 02/17/2019).

[9] P. Manickam, S. Sangeetha, and S. V. Subrahmanya. *Component-Oriented Development and Assembly: Paradigm, Principles, and Practice Using Java*. Boston, MA, USA: Auerbach Publications, 2013. ISBN: 9781466580992.

[5] J. Meyer. *JASMIN USER GUIDE*. 1996. URL: http://jasmin.sourceforge.net/guide.html (visited on 12/17/2019).

[10] O. Nierstrasz and T. D. Meijler. *Research Directions in Software Composition*. Tech. rep. 1995.

[11] O. Nierstrasz and D. Tsichritzis, eds. *Object-oriented Software Composition*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1995. ISBN: 0-13-220674-9.

[12] *OutSystems tools and components*. URL: https://success.outsystems.com/Evaluation/Architecture/1_OutSystems_Platform_tools_and_components (visited on 01/07/2019).

[13] *OutSystems tools and components - OutSystems*. URL: https://success.outsystems.com/Evaluation/Architecture/1_OutSystems_Platform_tools_and_components (visited on 01/29/2019).

[14] *Model-driven Development and Evolution of Customized User Interfaces*. EICS '13. London, United Kingdom: ACM, 2013, pp. 13–22. ISBN: 978-1-4503-2138-9. DOI: 10.1145/2494603.2480298. URL: http://doi.acm.org/10.1145/2494603.2480298.

[15] M. Potel. "MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java", Taligent Inc." In: (May 2011).

[16] J. R. Rymer. *The Forrester Wave™: Low-Code Development Platforms For AD&D Pros, Q4 2017*. Tech. rep. URL: https://www.outsystems.com/low-code-platforms (visited on 02/17/2019).

[17] *React – A JavaScript library for building user interfaces*. URL: https://reactjs.org/ (visited on 02/17/2019).

[18] *React Components, Elements, and Instances*. URL: reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html (visited on 09/19/2019).

[19] J. Sametinger. *Software Engineering with Reusable Components*. Berlin, Heidelberg: Springer-Verlag, 1997. ISBN: 3-540-62695-6.

[20] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201745720.

[21] *UI Flows - OutSystems*. URL: https://success.outsystems.com/Documentation/11/Developing{_}an{_}Application/Design{_}UI/Navigation/UI{_}Flows (visited on 09/20/2019).

[22] M. Wagner. *Genius Ways To Use Photoshop Smart Objects | Viget*. URL: https://www.viget.com/articles/smart-ways-to-use-adobe-photoshops-smart-objects/ (visited on 02/17/2019).

[23] A. J. A. Wang and K. Qian. *Component-Oriented Programming*. New York, NY, USA: Wiley-Interscience, 2005. ISBN: 0471644463.

[24] *What is Component Framework | IGI Global*. URL: https://www.igi-global.com/dictionary/component-framework/4912 (visited on 02/17/2019).

APPENDIX

Proof of concept

After designing the new model, a proof of concept allowed to validate the approach. The composition model proposed in this paper is an adaptation of the existing one in the OutSystems platform. It uses the React [17] library for compiling *widgets*, *web blocks* and screens to *React* components.

```
export interface AddressState {
   name: string,
   street: string,
   zipCode: string
}

export function Address (state: AddressState): JSX.Element  {
   (...)
   const [zipCode, setZipCode] = useState(state.zipCode);
   (...)
   return (
      <form>
         (...)
         <div>
            <span>ZipCode</span>
            <input id="zipCode" value={zipCode}
            onChange={handleZipCodeChange} />
         </div>
         (...)
      </form>
   );
}

```

Listing A.1: The component *Address* with the hook *zipCode* which is instantiated with the parameter passed in the state.

Listings A.1 and A.3 summarize the result of compiling the application described in Section 3. The *App* component represents the screen in Figure 5.3. This screen uses the

Address component. Data is kept within the scope of *App* (Listing A.3 lines 3-7) and is passed to *Address* as state (variables) (Listing A.3 line 12). The state is received by *Address* (Listing A.1 line 7) and is associated with a variable in its scope (Listing A.1 line 9), which in turn is associated with the *Input* element relative to the zip code (Listing A.1 line 16).

In the context of a purely textual editor, the customization of each component is done by introducing new properties by the developer himself. However, in the context of a visual editor it is necessary to represent in a structured and explicit manner which properties are subject to possible customizations, so that the editor recognizes which should allow the developer to be able to view and edit.

```
1  export interface AddressProps {
2     customizedWidgets: {
3        zipCode?: {
4           maxLength?:number
5        }
6     };
7  }
8  (...)
9  export function Address (state: AddressState): JSX.Element  {
10    (...)
11    const [zipCode, setZipCode] = useState(state.zipCode);
12    (...)
13    return (
14       <form>
15          (...)
16          <div>
17             <span>ZipCode</span>
18             <input id="zipCode"
19             maxLength={(
20             props.customizedWidgets != null and
21             props.customizedWidgets.zipCode != null and
22             props.customizedWidgets.zipCode.maxLength) or 5
23             value={zipCode}
24             onChange={handleZipCodeChange} />
25          </div>
26          (...)
27       </form>
28    );
29 }
```

Listing A.2: The component *Address* adapted to the new model.

In our proposal this is done by including a subelement (*customizedWidgets*) in the *props* of the component. By following this convention we are telling the graphic editor which *widgets* and properties can be edited for an instance of a *web block*. Listing A.2 illustrates changes to the *Address* component: a new structure for hosting new information (*AddressProps*) (lines 1-7) is declared. The property to modify is the character limit accepted by the *Input* element (*maxLength*), and it is optional to provide an alternative value for the property (lines 20-22). The next step is to associate these new fields with the corresponding elements. This requires checks to determine if the new fields are populated. Note that the *default* 5 value is not lost, which will be assigned if *maxLength* is not

populated.

In the *App* component in Listing A.4, a new *props* object is instantiated under the *AddressProps* type to pass customization to *Address* (lines 3-9).

If the *AddressProps* type does not have certain parameters (e.g., *size* of *Input*), these are not customizable, introducing a control over what properties can be changed from outside.

```
const App: React.FC = () => {

   const state : AddressState {
      name :"Joao Goncalves",
      street :"Rua Sem Nome",
      zipCode :"2222-222"
   };

   return (
      <div>
         <header>
            <Address {...state}/>
         </header>
      </div>
   );
};
```

Listing A.3: The component *App* that instantiates the component *Address* with the *state*.

```
const App: React.FC = () => {
   (...)
   const props : AddressProps = {
      customizedWidgets:{
         zipCode:{
            maxLength:10
         }
      }
   };

   return (
      <div className="App">
         <header className="App-header">
            <Address {...state} {...props}/>
         </header>
      </div>
   );
};
```

Listing A.4: The component *App* adapted to the new model.

Usability Testing: Script

One of the resources used to test if our approach was intuitive was a script. In the following pages, we present the script read by our two sample groups, Alfa and Beta.

It was requested to the participants to read the description of the application they would be interacting with, in order for them to understand the environment in which they would be faced with. After this, we requested them to follow a task according to their group.

UPPS Application

UPPS is an international delivery company which uses OutSystems to support their application.

The application created in Service Studio has a screen for the user to input their address data. The application owner wants to use the form in Figure 1 on multiple screens. The best plan of action for the developer assigned to this task would be to create a web block for the form and instantiate the web block in the screens the owner wants them.

Then, the developer creates the web block and instantiates them as needed. Meanwhile, a developer assigned to the portuguese department needs to use the web block. However, they need to change it, since the original only accepts zip codes of 5 characters.

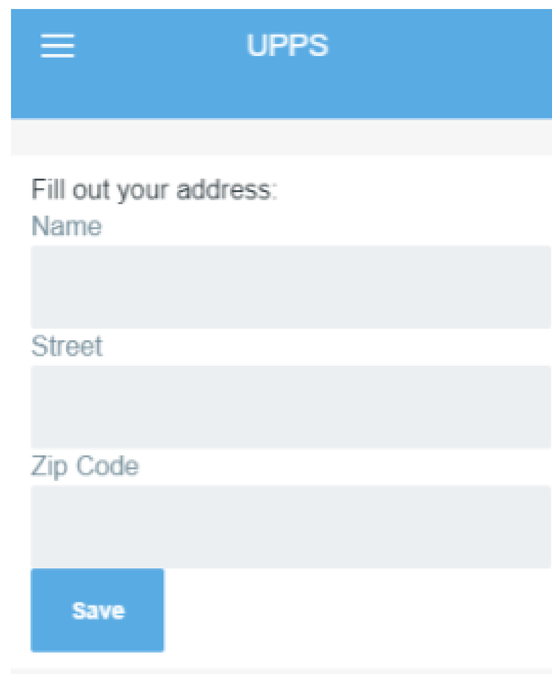


Figure 1: Address request

Beta group

Your task:

Change the web block Address instance in *PortugueseScreen* so that the *prompt* and *maxLength* correspond to the Portuguese standard:

prompt = "2222-222",
maxLength = 8,

without duplicating the web block or changing the *AmericanScreen*.

Note that changing the web block directly will affect the screens that instantiate it.

Alfa group

Your task:

Change the web block Address instance in *PortugueseScreen* so that the *prompt* and *maxLength* correspond to the Portuguese standard:

prompt = "2222-222",
maxLength = 8,

without accessing the web block definition or duplicating it.

Appendix

Usability Testing: Results

In total, the sample group for our usability test was comprised of 10 people. In table C.1, we show the results gathered from the Beta group. In table C.2, we show the results gathered from the Alfa group.

In table C.3, we show the questions asked to the participants to determine the System Usability Scale (SUS) score for each group approach.

Experience e (months)	**Question (rated from 1 to 5)**										**Time (mm:ss)**	**SUS (0-100)**
	I	**II**	**III**	**IV**	**V**	**VI**	**VII**	**VIII**	**IX**	**X**		
e <1	4	4	2	3	2	4	2	4	3	4	02:30	35
1 <e <6	1	4	3	2	2	3	3	4	2	2	05:00	40
1 <e <6	1	5	3	1	2	3	5	4	3	1	04:40	50
e <1	2	5	1	5	2	3	1	5	2	3	07:30	17,5
e >6	3	4	3	5	3	5	3	4	3	2	05:00	37,5

Table C.1: Beta group results.

Experience e (months)	**Question (rated from 1 to 5)**										**Time (mm:ss)**	**SUS (0-100)**
	I	**II**	**III**	**IV**	**V**	**VI**	**VII**	**VIII**	**IX**	**X**		
e <1	5	1	5	1	5	1	5	1	4	1	01:35	97,5
1 <e <6	4	1	5	1	5	1	5	1	4	1	00:55	95
1 <e <6	5	1	5	1	5	1	5	1	5	1	00:40	100
1 <e <6	5	1	5	2	5	1	5	1	4	1	02:20	95
e >6	5	1	5	1	4	1	5	1	3	3	00:40	87,5

Table C.2: Alfa group results.

I : I think that I would like to use this approach frequently.
II : I found this approach unnecessarily complex.
III : I thought this approach was easy to use.
IV : I think that I would need assistance to be able to use this approach.
V : I found the various functions in this approach were well integrated.
VI : I thought there was too much inconsistency in this approach.
VII : I would imagine that most people would learn to use this approach very quickly.
VIII : I found this approach very cumbersome/awkward to use.
IX : I felt very confident using this approach.
X : I needed to learn a lot of things before I could get going with this approach.

Table C.3: System Usability Scale questions.

Solution Comparison: Results

In table D.1 we present our analysis of the OutSystemsUIMobile module offered by OutSystems, with many UI patterns built in and ready to be used. The analysis is explained in Chapter 7.

The color scales used help showing widgets with higher values on a given index. By using color, it is easier to notice how many widgets we could influence positively on red color scales. Green color scales mean widgets offer more than original widgets. In this case, action parameters and preview are green since these are not offered in other widgets.

SW :	similar widgets
PP :	property parameters
AP :	action parameters
SP :	style parameters
PCW :	potential customizable widgets
V :	has preview

Table D.1: Indexes abbreviations.

Widgets	SW	PP	AP	SP	PCW	V
Columns	7	0	0	5	24	1
DisplayOnDevice	0	0	0	0	8	0.5
Gallery	0	0	5	0	10	0
MasterDetail	1	0	0	0	21	0
SplitScreen	1	0	2	0	7	0.5
AppFeedBack	0	0	0	0	21	0.5
Alert	0	3	0	1	16	1
BlankSlate	0	1	0	1	5	1
Card	3	1	0	1	2	1
CardAction	0	0	0	0	5	0.5
CardBackground	3	2	0	2	4	1
CardItem	3	0	0	0	5	0.5
CardSectioned	0	1	0	1	4	0.5
ChatMessage	0	3	0	1	21	1
FlipContent	0	0	2	1	10	0.5
FloatingContent	0	4	0	4	7	1
ListItemContent	3	0	0	0	5	0.5
Section	4	1	0	1	2	1
SExpandable	0	2	1	2	9	1
SectionGroup	1	0	1	1	1	0.5
Tag	0	1	0	1	1	1
ToolTip	0	0	1	2	13	0.5
UserAvatar	0	3	0	3	6	1
ActionSheet	0	0	1	0	17	0.5
Animate	0	0	0	3	1	0.5
AnimatedLabel	0	0	0	0	3	0.5
Carousel	0	0	8	3	9	0.5
Badge	1	2	0	2	2	1

Figure D.1: OutSystemsUI widgets results.

Short article

The following is a short article, summarizing this thesis most important aspects, submitted to INForum: Computer Science Conference, presented by Professor João Costa Seco at the 11th INForum - Simpósio de Informática, Department of Information Systems, University of Minho, 6/09/2019.

Composição e customização num modelo de componentes para interfaces gráficas

João Gonçalves[1], Hugo Lourenço[2], and João Costa Seco[1]

[1] NOVA LINCS - Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
[2] OutSystems, Portugal

Resumo As linguagens de programação visual são escolhidas para o desenvolvimento rápido de interfaces de utilizador devido à facilidade de uso, legibilidade, reusabilidade de componentes – *widgets* –, e uma pré-visualização imediata dos efeitos pretendidos. Contudo, os modelos mais comuns de composição para formar interfaces de utilizador são *black-box*: combinam *widgets* existentes para formar novos *widgets*, mas não permitem em geral a modificação indiscriminada dos seus componentes internos. Para os programadores conseguirem modificar internamente os elementos de uma composição e obterem uma pré-visualização dessa modificação, os modelos subjacentes têm que permitir receber e transmitir explicitamente propriedades do estado interno dos componentes para os elementos da composição.
Neste artigo apresentamos uma extensão da linguagem visual da plataforma *OutSystems* e um novo modelo de composição de componentes de interface de utilizador que permite a adaptação direta de elementos internos de composições de forma visual e interativa. Também apresentamos uma interação gráfica consistente com a linguagem visual, sem comprometer a experiência do programador. O modelo proposto é avaliado pela implementação de um protótipo completamente integrado na plataforma – ambiente de desenvolvimento visual e o seu compilador.

Palavras-chave: Plataformas de desenvolvimento *Low-code* · Linguagens de programação visuais · Modelo de composição *white-box* · Design baseado em componentes · Customização de componentes.

1 Introdução

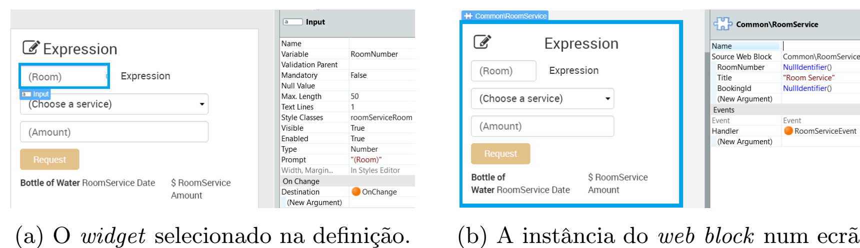
O desenvolvimento de aplicações com interface de utilizador para a Internet usando linguagens de programação visuais baseia-se habitualmente na utilização de abstrações visuais, ou componentes, que se traduzem em artefactos básicos ou compostos nas tecnologias subjacentes. Um modelo de componentes define as regras a utilizar na construção das interfaces e na geração e combinação das suas representações visuais. Num modelo uniforme, as composições são consideradas elas próprias componentes, que podem ser usadas noutras composições. Uma das vantagens deste paradigma é a reduzida ocorrência de erros devido à utilização de regras bem definidas de composição. No entanto, um modelo de composição que não prevê alterações na representação visual dos elementos de uma composição,

não facilita o desenvolvimento ou utilização de composições e requer muitas vezes a utilização de subterfúgios da linguagem como a passagem de parâmetros ou a manipulação das estruturas de suporte, e.g. folhas de estilo.

Num design baseado em componentes, a implementação de cada componente é encapsulada, não revelando detalhes de implementação e configuração ao contexto exterior (*black-box*) [1], permitindo apenas que os componentes se liguem através de conexões pré-determinadas. Contudo, num contexto visual, onde os elementos de uma composição são observáveis, a customização (controlada) de elementos da composição devia ser permitida, com um ganho significativo ao nível da experiência do programador. Esta situação motiva a definição de um novo modelo de composição que encapsula e parametriza um componente [2], e expõe uma seleção de propriedades dos seus elementos internos para customização. A esta motivação acrescenta-se o desenvolvimento de um editor visual capaz de configurar tais elementos e um compilador para produzir artefactos que transmitem e configuram as composições definidas.

Uma plataforma de desenvolvimento *Low-code* de aplicações *web* é um ambiente com uma linguagem de programação visual que depende de métodos declarativos e visuais com reduzida necessidade para código manuscrito. Apesar de disporem de uma grande oferta de componentes representativos de recursos encontrados em aplicações web (e.g., elementos *HTML*), as alterações na interface de utilizador que não sejam suportadas apenas o podem ser através de customização manual [7] – esta responsabilidade deve recair no programador. Se em vários sítios se observar na interface de utilizador um conjunto de componentes próximos com uma lógica semelhante que os interliga (e.g., formulários), a representação única do conjunto numa só composição motivaria a reutilização de recursos e um desenvolvimento mais rápido.

As plataformas *Low-code* oferecem ferramentas de customização de componentes, suportadas por editores onde os programadores compõem ecrãs ao arrastar e largar elementos visuais. A *OutSystems* oferece um *IDE* deste género, o *Service Studio*, com *widgets* que permitem a definição de ecrãs e *web blocks* (conjuntos de *widgets* reutilizáveis).



(a) O *widget* selecionado na definição. (b) A instância do *web block* num ecrã.

Figura 1: A experiência associada a *web blocks*.

Na Figura 1a apresenta-se a visão que um programador tem ao definir um *web block*. Nesta visão as propriedades de todas as *widgets* estão disponíveis e podem ser editadas. Nesta figura está selecionado um *widget input* (caixa azul), e, no editor à direita, constam as suas propriedades que podem ser modificadas.

A Figura 1b ilustra o uso do *web block* na construção de um ecrã. Uma instância de um *web block* é tratada, neste contexto, como uma *black-box* do ponto de vista de edição: não é possível selecionar *widgets* individuais do mesmo, mas apenas a instância como um todo (caixa grande azul). No entanto, note-se que um *web block* não é uma *black-box* do ponto de vista da visualização: seguindo uma abordagem *WYSIWYG* (*What You See Is What You Get*), o conteúdo do mesmo contribui para a pré-visualização do conteúdo do ecrã.

Portanto, seria natural (e até expectável) interagir com o conteúdo do *web block* apresentado no ecrã, alterando diretamente as propriedades dos *widgets*. Tal não é permitido, de momento, pela plataforma *OutSystems*. As eventuais customizações dos *widgets* obrigam a trabalho extra, tanto por parte do programador que cria o *web block*, como do programador que o usa. Por exemplo, é possível usar parâmetros de entrada para fornecer valores alternativos às propriedades dos *widgets* contidos no *web block*. No entanto, esta alternativa não é compatível com uma abordagem *WYSIWYG*, uma vez que não é possível determinar em tempo de edição o valor dos parâmetros, perdendo assim a pré-visualização instantânea.

As linguagens de programação visuais destacam-se pela sua facilidade de uso, oferecendo a quem não tem conhecimentos especializados em programação tradicional ferramentas para criar aplicações ricas. Assim, os programadores que usam estas linguagens conseguem criar e distribuir aplicações em menos tempo. No entanto, é difícil criar uma linguagem visual que substitua todos os recursos que uma linguagem textual oferece. Neste trabalho é discutida a falta de customização direta em composições de *widgets*, algo trivial de se concretizar numa linguagem textual. A motivação para este trabalho resulta de uma maior facilidade de uso para este tipo de plataformas através de um novo modelo de composição, em que é expectável acelerar ainda mais o desenvolvimento de aplicações.

Figura 2: A aplicação de entregas ao domicílio com o *web block* do formulário que aceita moradas.

Um caso de uso possível relevante para este problema podia ser o seguinte: numa aplicação de entregas ao domicílio desenvolvida na plataforma *OutSystems*, é necessário requisitar a morada do utilizador. Faz sentido que o formulário na

Figura 2 seja um *web block*, pois é usado em mais que um ecrã. Contudo, este é desenhado tendo em conta a morada estadunidense, que tem um código postal de cinco dígitos, enquanto que a portuguesa tem sete. Nesta situação, a propriedade relativa ao tamanho máximo de carateres (*maxLength*) deve ser customizável, tal como o criador do bloco deve poder decidir se tal é permitido.

Assim, o problema pode ser dividido em duas vertentes: que propriedades devem ser expostas ao exterior nativamente pela plataforma e de que maneira estas podem ser permitidas a customizar. Com este problema resolvido, os criadores conseguem criar *web blocks* mais genéricos, preparados para mais situações, em que os programadores possam customizá-los de acordo com o contexto em que os blocos se inserem, resultando num desenvolvimento mais acelerado e numa melhor utilização de recursos.

Os objetivos deste trabalho são apresentar um modelo de composição de elementos de interface de utilizador na plataforma *OutSystems*, com os mesmos padrões que qualquer outro componente (reusabilidade e pré-visualização instantânea), e, consequentemente, as alterações feitas à linguagem visual para ter um protótipo que se estendesse entre todas as vertentes da plataforma. Este modelo permite composições de componentes *white-box*, com acesso à instância e propriedades dos componentes interiores. Para isto, o modelo existente no *Service Studio* foi adaptado para guardar informação relativa a que propriedades são customizáveis e os seus potenciais valores, tal como a linguagem visual foi alterada para expor e permitir a customização de propriedades. Para além disto, o compilador foi igualmente alterado para criar aplicações finais que suportassem estas modificações.

Espera-se que este trabalho contribua com novas formas de composição de elementos visuais que, num modelo de composição, possam fornecer uma representação visual fiel à sua implementação, proporcionando uma experiência intuitiva e satisfatória ao programador.

Nas próximas secções, será dada uma breve introdução à plataforma *OutSystems* (2), com definições pertinentes a este trabalho, seguindo-se o modelo proposto (3), alterações à linguagem visual da plataforma (4) e ao compilador (5). De seguida, relacionamos o nosso trabalho com o de outros autores (6) e projetamos algumas direções para trabalho futuro (7).

2 Plataforma OutSystems

A *OutSystems* é uma das fundadoras do mercado de plataformas de desenvolvimento *Low-code* [4], fornecendo modelos visuais que dependem de uma linguagem de abstração mais alta, fácil de aprender. Permite aos programadores desenvolverem sem a preocupação de escrever código defeituoso e criar aplicações web e móveis rápidas, seguras e altamente escaláveis. A plataforma abrange todas as etapas do ciclo de vida das aplicações, com ambientes de desenvolvimento e implementação [5]. O programador interage com estes para criar as suas aplicações na linguagem *OutSystems* que são posteriormente compiladas e implementadas no servidor da plataforma.

O *Service Studio* é o ambiente de desenvolvimento em que os programadores criam a interface de utilizador das suas aplicações ao arrastar e soltar elementos visuais numa tela. Para garantir o desempenho, o ambiente desloca o trabalho do programador para o *Platform Server*, enviando uma versão compactada da aplicação [5]. O *Platform Server* lida com todas as tarefas relacionadas à compilação e à distribuição de aplicações. Gera, otimiza, compila e implementa um aplicação *OutSystems* num servidor de aplicações web padrão.

2.1 Espaço de trabalho

Os módulos do *Service Studio* permitem estruturar aplicações em partes com finalidades diferentes. A área do editor tem várias consolas diferentes, que ajudam a editar a tela, conforme a Figura 3.

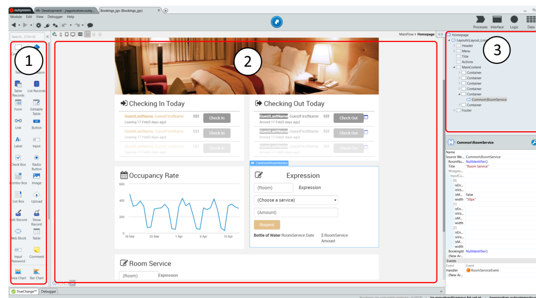


Figura 3: Espaço de trabalho do *Service Studio*
1: Caixa de ferramentas de *widgets* 2: Tela 3: Árvore de *widgets*

A tela é onde a visualização do módulo acontece e, em tempo real, mostra o aspecto de um ecrã ou *web block*. Aqui, é mostrado como os *widgets* serão renderizados no produto final, tal como é possível interagir para se adicionar ou remover elementos e selecionar cada um apresentado na tela para editar as suas propriedades.

À direita, a área da árvore pode mostrar a interface do módulo (lógica da aplicação) ou a composição de uma tela. Quando um elemento visual é selecionado, a área das propriedades aparece abaixo da árvore de elementos, onde o programador pode editá-las.

Os ecrãs são as páginas da interface de utilizador com as quais os utilizadores finais interagem. Estes podem ser compostos por *widgets* ou *web blocks*. A composição é uma árvore de elementos, semelhante à estrutura de uma *markup language*, que pode ser alterada ao arrastar componentes de dentro ou fora de outros escopos de componentes. Também podem ter as suas ações (lógica que pode ser ativada pelo utilizador), parâmetros de entrada e variáveis locais.

A caixa de ferramentas de *widgets* contém todos os blocos de construção nativos fornecidos pelo *Service Studio*. Assim, para criar um formulário num

ecrã, o programador arrasta o *widget* Formulário para a tela e preenche as propriedades obrigatórias para que este funcione. O *widget Input* permite que os programadores estendam os formulários ainda mais com mais um espaço para os utilizadores finais preencherem. Este, como os outros, também tem as suas propriedades, que podem ser *runTimeProperies*, propriedades que são apenas avaliadas ao correr a versão compilada, ou *designTimeProperties*, propriedades simples que determinam como o *widget* será renderizado na tela e na versão final.

Para criar conjuntos de *widgets* reutilizáveis (podem ser usados mais que uma vez em ecrãs e contextos diferentes) existem os *web blocks*. Tal como um ecrã, um *web block* pode ter as suas próprias ações, lógica, parâmetros de entrada e saída, e variáveis locais. Os *web blocks* podem encapsular vários *widgets*, o que significa que podem ser *widgets* nativos ou outros *web blocks*. Ao selecionar uma instância de um *web block*, o programador não tem acesso aos seus componentes.

Os *widgets* nativos permitem que os programadores acedam e modifiquem as suas propriedades, tal como numa linguagem de programação textual. A customização dentro de um ecrã ou num *web block* é a mesma: os programadores podem aceder às mesmas propriedades nas duas situações. Numa aplicação com várias instâncias do mesmo *web block*, existe a necessidade de customizar de maneira diferente cada uma , de acordo com o contexto em que se inserem. No entanto, o acesso aos componentes internos é restrito pela plataforma.

Existem alternativas para resolver este problema, mas implicam vários compromissos. A duplicação de *web blocks* oferece implementações mais rápidas, mas muito tempo é desperdiçado cada vez que o programador alterar um dos duplicados, pois terá que alterar todos os outros para manter a consistência. Usar parâmetros de entrada garante reusabilidade, mas também é demoroso tornar todas as propriedades customizáveis. Como estes são apenas avaliados em *run time*, a pré-visualização instantânea é comprometida. Por fim, as folhas de estilo permitem customizar *web blocks* muito rapidamente e com uma pré-visualização fiel às alterações, mas para além de necessitarem de conhecimento especializado, apenas funcionam na customização do estilo.

3 Modelo proposto

O modelo de composição proposto neste trabalho é uma adaptação do já existente na plataforma *OutSystems*. Esta usa a biblioteca *React* [3] para compilação de *widgets*, *web blocks* e ecrãs para componentes *React*. Nas Listagens 1.1 e 1.3 apresenta-se de forma resumida o resultado da compilação da aplicação descrita na Secção 1. O componente *App* representa o ecrã na Figura 2. Este ecrã usa o componente *Address*. Os dados são mantidos no escopo da *App* (Listagem 1.3 linhas 3-7) e são passados ao *Address* como estado (variáveis) (Listagem 1.3 linha 12). O estado é recebido por *Address* (Listagem 1.1 linha 7) e é associado a uma variável no seu escopo (Listagem 1.1 linha 9), que por sua vez é associado ao elemento *Input* relativo ao código postal (Listagem 1.1 linha 16).

No contexto de um editor puramente textual, a customização de cada componente é feita através da introdução de novas propriedades pelo próprio pro-

gramador. No entanto, no contexto de um editor visual é necessário representar de forma estruturada e explícita quais as propriedades sujeitas a possíveis customizações, por forma a que o editor reconheça quais deve permitir que o programador possa ver e editar.

```
1  export interface AddressState {
2      name: string,
3      street: string,
4      zipCode: string
5  }
6
7  export function Address (state: AddressState): JSX.Element  {
8      (...)
9      const [zipCode, setZipCode] = useState(state.zipCode);
10     (...)
11     return (
12         <form>
13             (...)
14             <div>
15                 <span>ZipCode</span>
16                 <input id="zipCode" value={zipCode}
17                 onChange={handleZipCodeChange} />
18             </div>
19             (...)
20         </form>
21     );
22 }
```

Listagem 1.1: O Componente *Address* com o *hook zipCode* que é instanciado com o parâmetro passado no estado.

```
1  export interface AddressProps {
2          customizedWidgets: {
3              zipCode?: {
4              maxLength?:number
5          }
6      };
7  }
8  (...)
9  export function Address (state: AddressState): JSX.Element  {
10     (...)
11     const [zipCode, setZipCode] = useState(state.zipCode);
12     (...)
13     return (
14         <form>
15             (...)
16                 <div>
17                     <span>ZipCode</span>
18                         <input id="zipCode"
19                             maxLength={(
20                                 props.customizedWidgets != null and
21                                 props.customizedWidgets.zipCode != null and
22                                 props.customizedWidgets.zipCode.maxLength) or 5
23                                 value={zipCode}
24                      onChange={handleZipCodeChange} />
25                 </div>
26             (...)
27         </form>
28     );
29 }
```

Listagem 1.2: O Componente *Address* adaptado ao novo modelo.

Na nossa proposta tal é feito através da inclusão de um subelemento (*customizedWidgets*) nas *props* do componente. Ao seguirmos esta convenção estamos

a indicar ao editor gráfico quais as *widgets* e propriedades que podem ser editadas para uma instância de um *web block*. Na Listagem 1.2 ilustram-se as alterações no componente *Address*: é declarada uma nova estrutura para albergar a nova informação (*AddressProps*) (linhas 1-7). A propriedade a modificar é o limite de caracteres aceites pelo elemento *Input* (*maxLength*), sendo opcional fornecer um valor alternativo para a propriedade (linhas 20-22). O próximo passo é associar estes novos campos aos elementos correspondentes. Para isto, são necessárias verificações para determinar se os novos campos estão preenchidos. Note-se, que não se perdeu o valor *default* *5*, que será atribuído caso *maxLength* não esteja populado.

No componente *App* na Listagem 1.4, um novo objeto *props* é instanciado sob o tipo *AddressProps* para transmitir a customização ao *Address* (linhas 3-9). Se o tipo *AddressProps* não tiver certos parâmetros (e.g., *size* do *Input*), estes não são customizáveis, introduzindo um controle sobre que propriedades podem ser alteradas pelo exterior.

```
1  const App: React.FC = () => {
2
3      const state : AddressState {
4          name :"Joao Goncalves",
5          street :"Rua Sem Nome",
6          zipCode :"2222-222"
7      };
8
9      return (
10         <div>
11             <header>
12             <Address {...state}/>
13             </header>
14         </div>
15     );
16 };
```

Listagem 1.3: O Componente *App* que instancia o componente *Address* com o estado *state*.

```
1  const App: React.FC = () => {
2      (...)
3      const props : AddressProps = {
4          customizedWidgets:{
5              zipCode:{
6                  maxLength:10
7              }
8          }
9      };
10
11     return (
12         <div className="App">
13             <header className="App-header">
14             <Address {...state} {...props}/>
15             </header>
16         </div>
17     );
18 };
```

Listagem 1.4: O Componente *App* adaptado ao novo modelo.

3.1 Alterações ao modelo da *OutSystems*

O modelo proposto na secção anterior é um produto do modelo de composição que nós adaptámos na linguagem da *OutSystems*. Neste modelo, os *web blocks* têm uma classe que os define, e outra para a sua instância.

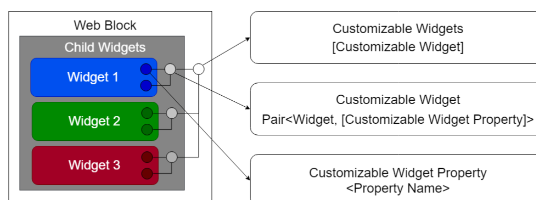


Figura 4: Novo modelo para a definição de blocos reutilizáveis.

Na classe que define um *web block* foi adicionada uma coleção a indicar que propriedades podem ser customizadas pelo exterior (*Customizable Widgets*, Figura 4) - apenas as propriedades presentes podem ser alteradas. Aqui guarda-se uma nova estrutura, *CustomizableWidget*, que guarda as propriedades customizáveis - apenas o nome é necessário - e o *widget* pai, para efeitos de procura.

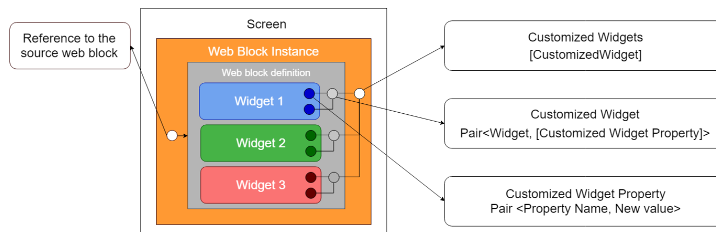


Figura 5: Novo modelo para a instanciação de blocos reutilizáveis.

A instância do *web block* tem propriedades distintas de outras instâncias do mesmo *web block*, pelo que tem a sua própria classe para representá-la. No entanto, a representação visual da composição é mantida na definição, sendo necessário guardar os valores para customização na instância. Nesta, introduziu-se a coleção *Customized Widgets* (Figura 5) que guarda os *widgets* a customizar e as respetivas alterações do programador.

Neste momento, o modelo está preparado para *design-time properties*, mas não para *runtime properties*, porque a customização é guardada como *string*. Como próximo passo, pretendemos usar expressões, para estender a customização a todas as propriedades. Desta forma também seria possível introduzir dependências entre widgets. Este problema é interessante, porque a passagem de

informação entre componentes cria novas oportunidades: um componente herdar propriedades de outro automaticamente (e.g., largura); um programador ter acesso ao modelo de dados de um componente e poder inseri-lo noutro (e.g., um componente partilhar o acesso aos dados de um utilizador a outro componente diminuiria acessos redundantes).

4 Interação com o modelo

Definidas as novas estruturas para que a customização seja permitida pelo modelo de composição, a geração das representações visuais foi alterada. Tal como na Listagem 1.2, as estruturas são verificadas por potenciais customizações.

A interação no *Service Studio* opera segundo uma arquitetura *Model-View-Presenter* (Figura 6), em que o programador interage com a tela (*view*), que despoleta um evento, evento esse que é subscrito pelo *presenter*, que usa um comando para modificar o *model*. O *presenter* usa um contador de gerações para atualizar as partes da tela desatualizadas: uma alteração num objeto do modelo incrementa o contador correspondente que faz com que a tela seja atualizada. Assim, foram criados comandos para tornar as propriedades de *widgets* customizáveis em *web blocks*, tal como customizar certas propriedades de certas instâncias. Neste momento, conseguimos apenas tornar todas as propriedades de todos os *widgets* customizáveis e alterar uma propriedade num dado *widget* de um dado *web block*.

Para que as representações visuais das partes customizadas sejam atualizadas, foi necessário que as novas estruturas introduzidas no modelo incrementassem o contador respetivo no *presenter*. O passo seguinte foi alterar o *HtmlGenerator*, responsável por gerar o código *HTML* mostrado na tela, para o programador obter uma representação visual fiel às suas alterações. Ao verificar cada *widget* que compõe um *web block*, são procuradas possíveis customizações nas novas estruturas. Se houver, a propriedade correspondente recebe a customização em vez do valor *default*.

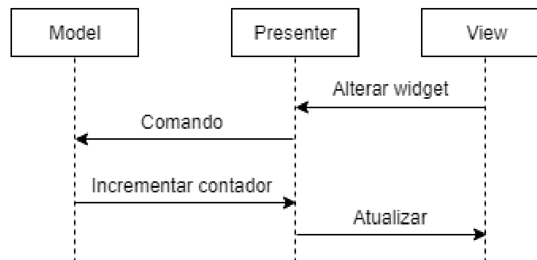


Figura 6: Linha do tempo quando o programador altera um *widget*.

5 Compilação

Tendo em conta que nenhum dos *widgets* foi alterado, só em *run time* é que é possível associar os valores do novo modelo às representações visuais de cada componente. Assim, após criar as novas estruturas, o compilador foi adaptado para ser compatível com o novo modelo, pelo que é necessário verificar se existem customizações pendentes e gerar um modelo semelhante ao apresentado na Secção 1, para que em *run time* se possa escolher qual dos valores é usado: o valor novo ou o *default*.

As alterações consistiram em criar interfaces que unissem as estruturas usadas no *Service Studio* às do compilador, de forma a que fosse possível aceder às novas coleções criadas. De seguida, o gerador de código *JavaScript* foi alterado para procurar por propriedades customizáveis - caso o sejam, é gerado código semelhante ao apresentado na Listagem 1.2, que verifica se de facto existem customizações a serem transmitidas ao *web block* - e por customizações pendentes, onde é gerado código semelhante à Listagem 1.4.

6 Trabalho relacionado

O trabalho aqui realizado consiste em uniformizar uma linguagem visual para que esta não seja apenas mais fácil de usar, mas também atinja os níveis mais altos de qualidade. O suporte para a variabilidade no desenvolvimento de interfaces de utilizador é importante, porque permite criar produtos mais específicos às necessidades do utilizador. A customização de interfaces de utilizador ajuda, neste aspeto, a criar produtos com mais valor. Tendo em conta que a customização de *widgets* é manualmente inserida pelo programador, a customização de composições deve seguir a mesma regra, para que todos os casos de uso continuem a ser possíveis. Para isto, usámos o paradigma da parametrização para transmitir os novos valores às propriedades dos *widgets* a customizar.

Usar parametrização como base para resolver o problema aqui apresentado não é novo. Lizcano et al [6] introduziram um modelo de composição para componentes de interface de utilizador seguindo um paradigma de parametrização. As vistas dos componentes são *black-box*, pelo que futuras modificações apenas poderão ser feitas através de parâmetros. Assim, é aconselhado criar componentes já preparados para eventuais customizações.

De igual forma, Pleuss et al [7] introduzem um novo conceito, Família *UI*, que consiste em ecrãs completos para responder aos requisitos dos utilizadores, que tendem a usar ecrãs semelhantes nas suas aplicações. Mas, utilizadores podem requisitar casos de uso imprevistos pela implementação da interface de utilizador. Neste caso, é necessário que o utilizador introduza a customização manualmente. Para isto, Pleuss et al usam modelos associados à interface de utilizador para a customizar. Apesar da motivação ser diferente – customização de ecrãs em vez de conjuntos de *widgets* –, os conceitos são os mesmos com os que são apresentados neste trabalho: são adicionados novos modelos onde se pode guardar a customização que o programador pretende fazer ao objeto, quer seja um ecrã ou um bloco.

7 Conclusões e trabalho futuro

Este trabalho apresenta um novo modelo de composição inspirado no modelo presente da plataforma *OutSystems*. O novo modelo permitirá um desenvolvimento de aplicações ainda mais rápido do que o atual, graças a uma interação mais uniforme que oferece ao programador o poder de customizar as suas próprias composições de *widgets*. O protótipo resultante prova que é possível customizar componentes dentro de uma composição através de um paradigma de parametrização, sem comprometer a pré-visualização das alterações e sem perder a unicidade da definição de um bloco.

Como trabalho futuro, é esperado construir uma interação mais compreensiva, atendendo às necessidades dos criadores e utilizadores de composições, para que seja possível determinar que propriedades são customizáveis e customizar apenas as propriedades que o são, havendo um controle assegurado não só pelos criadores mas como pela plataforma. Para além disto, customizações com *runTimeProperties* permitiriam situações interessantes a explorar, como discutido na Secção 2.

Referências

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
2. Oscar Nierstrasz and Dennis Tsichritzis (Eds.). 1995. Object-Oriented Software Composition. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.
3. React – A JavaScript library for building user interfaces, URL: `https://reactjs.org/` (visitado em 2019-07-11).
4. John R. Rymer, The Forrester Wave™: Low-Code Development Platforms For AD&D Pros, Q4 2017, URL: `https://www.outsystems.com/low-code-platforms` (visitado em 2019-07-11).
5. OutSystems tools and components - OutSystems, URL: `https://success.outsystems.com/Evaluation/Architecture/1_OutSystems_Platform_tools_and_components` (visitado em 2019-07-11).
6. David Lizcano, Fernando Alonso, Javier Soriano, and Genoveva López. 2011. A new end-user composition model to empower knowledge workers to develop rich internet applications. J. Web Eng. 10, 3 (September 2011), 197-233.
7. Andreas Pleuss, Stefan Wollny, and Goetz Botterweck. 2013. Model-driven development and evolution of customized user interfaces. In Proc. 5th ACM SIGCHI symposium on Engineering interactive computing systems (EICS '13). ACM, New York, NY, USA, 13-22.