**António Simões Ferreira**

Bachelor in Computer Science

# Integration of Visual Languages with SCS tools in the Software Development Industry

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

Master of Science in
**Computer Science and Informatics Engineering**

| | |
|---|---|
| Advisers: | Jácome Cunha, Assistant Professor, University of Minho |
| | André Pinto Vieira, Software Engineer, OutSystems |
| Co-advisers: | João Ricardo Viegas da Costa Seco, Assistant Professor, FCT-UNL |
| | Hugo Lourenço, Software Engineer, OutSystems |

Examination Committee

| | |
|---|---|
| Chairperson: | Miguel Jorge Tavares Pessoa Monteiro, Assistant Professor, FCT-UNL |
| Raporteur: | João Paulo Fernandes, Assistant Professor, University of Coimbra |
| Members: | Jácome Cunha, Assistant Professor, University of Minho |
| | João Ricardo Viegas da Costa Seco, Assistant Professor, FCT-UNL |
| | André Pinto Vieira, Software Engineer, OutSystems |

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
**UNIVERSIDADE NOVA** DE LISBOA

**September, 2019**

**Integration of Visual Languages with SCS tools in the Software Development Industry**

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 start by thanking the Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa (FCT-UNL), in particular to the Informatics Department, for providing me with all the necessary educational tools to start off my professional career. Also I would like to thank OutSystems for providing a scholarship for this dissertation. A big thanks to my advisers, Jácome Cunha, André Vieira, João Seco and Hugo Lourenço for mentoring me in this thesis. Their advice was really helpful and allowed me to constantly improve my work.

Also, a big thanks to all of my OutSystems friends: Francisco Cunha, David Mendes, Lara Borisoglebski, Joana Tavares, Miguel Loureiro, João Gonçalves, Miguel Madeira, Francisco Magalhães, Michael Silva, Nuno Calejo, Mariana Cabeda and Nuno Pulido, for all of the great and fun lunches, coffee breaks and conversations over the pool table.

Most importantly a huge thank you to my parents. Their unconditional support was vital and one of the main reasons why I was always capable of withstand the hard times in my journey and be where I am today. Also, a big thanks to my brother, for all of the times he distracted me from my work, which allowed me to clear off my mind for bit. Thank you Bolota for being the fluffy ball of fur that always wrecked havoc when running around the house.

And lastly, a big thanks to all of my close friends, whose names don't need to be mentioned, because they know who they are. All of their support, conversations, and fun times we spent together is something that I am and always will be grateful for.

*“Once I did bad and that I heard ever. Twice I did good, but that I heard never.”*

*-Dale Carnegie*

# Abstract

Source Control Systems (SCS), also known as Version Control Systems (VCS), help teams to organize and track changes in the software development process. These systems have become vital in the software development industry as the increased growth and geographical diversity of teams, forced them to find solutions to deal with multiple people accessing the same pieces of software concurrently. Although for a while SCS seemed to be coping well with the needs of software development, the rise of the low-code platforms and Visual Programming Languages (VPLs) brought a new challenge to version control: how to manage visual artifacts without losing SCS functionalities? The biggest cause of this challenge is the fact that SCS are mostly oriented to work with text-based programming languages. Thus text-oriented SCS are (in general) incapable of dealing with visual artifacts as well as they do with text. So, to cope with the loss of SCS functionalities in VPLs projects, teams either accept and work with this loss or are forced to come up with a solution of their own to tackle a specific version control problem. These issues can be found in the OutSystems platform, which is our case study.

To solve this problem, we propose a system, termed OSGit, that acts as a *man-in-the-middle* between the low-code platform and the designated SCS. The proposed system will translate the requested version control operations from the low-code platform to native operations of the given SCS. In operations that require visual artifacts to be managed - like applying blames - we propose the creation of metadata files. The metadata files contain the needed information about the visual elements used to build applications in the low-code platform. This information is thus a compacted representation of the visual elements through text. Therefore, using metadata files will allow the SCS to correctly handle the required visual artifacts while integrating this system with a low-code platform or a VPL.

The produced system obtained auspicious results in the usability tests that were performed and that featured ten OutSystems developers. They showed great satisfaction when using OSGit and also gave suggestions for future improvements. OSGit bridges the gap between text-based SCS and visual artifacts, which proves the possibility of integrating these systems in the world of VPL with a high-level of user satisfaction.

---

**Keywords:** Source Control Systems, Version Control Systems, Visual Programming Languages, Low-Code, OutSystems

# Resumo

Os Source Control Systems (SCS), também conhecidos como Version Control Systems (VCS), ajudam equipas a organizar e a localizar mudanças no código de um software durante a sua construção. Estes sistemas tornaram-se vitais na indústria de desenvolvimento de software que, devido ao crescimento e à diversidade geográfica das equipas, levaram-nas a procurar soluções para lidar a existência de várias pessoas a trabalhar concorrentemente no mesmo projecto. Contudo, a ascensão das plataformas *low-code* e das linguagens de programação visuais, trouxeram um novo desafio às metodologias de controlo de versões: como gerir artefatos visuais sem perder as funcionalidades dos SCS? Este desafio emerge com o facto da maioria dos SCS fornecerem apenas funcionalidades para software construído apartir de linguagens de programação baseadas em texto. Agora que a integração e o uso das linguagens de programação visuais na indústria de desenvolvimento de software emergiram, os SCS desenvolvidos para texto, são (no geral) incapazes de lidar com os artefatos visuais da mesma forma que o fazem com elementos textuais. Para lidar com a perda de funcionalidades dos SCS em projectos que usem VPLs, as equipas podem optar por duas vertentes: aceitar as perdas; ou serem forçadas a criar uma solução para lidar com o problema de controlo de versões em artefactos visuais.

Para resolver este problema, propomos um sistema, denominado OSGit, que atua como um *man-in-the-middle* entre a plataforma da OutSystems e o SCS escolhido. Este sistema traduzirá as operações de controlo de versões usadas na plataforma para operações nativas do SCS escolhido. Em operações que exigem que artefatos visuais sejam manipulados, propomos a criação de ficheiros de meta dados. Estes ficheiros contêm informações sobre elementos visuais usados para criar aplicações em OutSystems. Estes meta dados são então uma representação destes elementos através de texto. Desta forma o SCS manipulará corretamente os artefatos visuais presentes na plataforma.

O sistema proposto obteve resultados bastante favoráveis nos testes de usabilidade realizados. Nestes testes participaram dez programadores da OutSystems. Estes participantes mostraram grande satisfação em relação ao OSGit, dando também sugestões para melhorias do mesmo. O OSGit colmata assim, a lacuna entre o SCS e os artefatos visuais, o que comprova a possibilidade de integrar estes sistemas no mundo das VPLs com um alto nível de satisfação dos utilizadores.

**Palavras-chave:** Sistemas de Controlo de Versões, Linguagens de Programação Visuais, Low-Code, OutSystems

# Contents

**List of Figures** **xvii**

**List of Tables** **xix**

**Listings** **xxi**

**Acronyms** **xxiii**

**1 Introduction** **1**

1.1 Motivation . . . . . 2

1.2 Proposed Solution . . . . . 3

1.3 Key Contributions . . . . . 4

1.4 Structure . . . . . 5

**2 Low-Code Software Development** **7**

2.1 Low-Code Software Development . . . . . 7

2.2 OutSystems . . . . . 7

2.2.1 Service Studio . . . . . 8

2.2.2 The OutSystems Application Files . . . . . 10

**3 The Different Shapes of SCS** **11**

3.1 Source Control Systems . . . . . 11

3.2 Source Control in Low-Code Platforms . . . . . 11

3.2.1 Versioning in OutSystems . . . . . 12

3.2.2 Versioning in Mendix . . . . . 14

3.3 Source Control in IDEs . . . . . 15

3.3.1 Source Control in Unreal Engine 4 . . . . . 15

3.3.2 Source Control in Unity . . . . . 16

3.3.3 Summary . . . . . 16

3.4 Source Control in Modeling Languages . . . . . 17

3.4.1 Summary . . . . . 18

3.5 Source Control in Spreadsheets . . . . . 18

3.6 Source Control in Text-Based Languages . . . . . 19

3.7 Choosing the SCS to integrate with OutSystems . . . . . 21

3.7.1 Mercurial Vs. Git . . . . . 22
3.7.2 The Chosen Git Operations . . . . . 23

**4 Integrating Git with OutSystems 25**
4.1 Overview . . . . . 25
4.2 Representing Visual Objects as Text . . . . . 26
4.2.1 Generating the Metadata Files . . . . . 28
4.2.2 Deleting an Element . . . . . 30
4.2.3 The Metadata Directory . . . . . 31
4.3 The Settings Screen . . . . . 31
4.3.1 The Account Configuration Box . . . . . 32
4.3.2 The Git Configuration Box . . . . . 33
4.3.3 The Repository Configuration Box . . . . . 34
4.3.4 Saving settings . . . . . 34
4.4 The Commit Operation . . . . . 35
4.4.1 The Operation Implementation . . . . . 37
4.4.2 The Repository State . . . . . 39
4.5 The Blame Operation . . . . . 40
4.5.1 The Operation Implementation . . . . . 42
4.6 Additional Operations . . . . . 44
4.6.1 The Fetch Operation . . . . . 44
4.6.2 The History Operation . . . . . 45

**5 Evaluation 47**
5.1 Experiment Design . . . . . 47
5.2 Participants . . . . . 48
5.3 Execution . . . . . 49
5.4 Results . . . . . 49
5.4.1 The SUS . . . . . 53
5.5 Analysis . . . . . 54

**6 Conclusions 57**
6.1 Final Observations . . . . . 57
6.2 Future Work . . . . . 58

**Bibliography 61**

**A Post-Test Questionnaire 65**

**B SCS User Acceptance Exercise 71**

**C SCS User Acceptance Exercise 73**

**D SCS workflows in IDE's** **77**

# List of Figures

1.1 Schema of the current state of version control in the OutSystems platform. . 2

1.2 Schema of the proposed solution that establishes the communication between the OutSystems Service Studio, the chosen SCS and its external repository. . 4

2.1 The OutSystems Platform components, with an emphasis on the Service Studio and the Platform Server [38]. . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2 Service Studio overview while editing an application's User Interface (UI). . 9

2.3 Service Studio overview while editing an application's logic. . . . . . . . . . . 9

2.4 Service Studio overview while editing an application's database. . . . . . . 10

3.1 Merge situation, where the developer can visually compare two versions and merge them manually [26]. . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.2 Merge conflict situation between two versions of a Cascading Style Sheets (CSS) file [9]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.3 Merge conflict situation between two versions of an action [9]. . . . . . . . 13

3.4 Number of repositories of each SCS as of July 2019 [10]. . . . . . . . . . . . 22

4.1 Detailed schema of the proposed solution that establishes the communication between the Service Studio and a GitHub repository. . . . . . . . . . . . . . 25

4.2 The ESpace tree elements of the interface that form the *Bookings* application. 27

4.3 An example of the metadata information that is stored for each ESpace tree element. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.4 The web screens metadata file, where all application's web screens metadata is stored. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

4.5 The user actions metadata file content. . . . . . . . . . . . . . . . . . . . . . 30

4.6 The user actions metadata file content, with a deleted metadata entry. . . . . 30

4.7 An example of the generated metadata files directory. . . . . . . . . . . . . . 31

4.8 The mockup of the Settings menu, from where the users can configure the Git repository and authenticate to GitHub. . . . . . . . . . . . . . . . . . . . . 32

4.9 The *Git Credential Manager* GitHub authentication window. . . . . . . . . . 33

4.10 The implemented version of the Settings Screen UI. . . . . . . . . . . . . . . . 35

4.11 The commit message input, for the user to introduce his commit message. . . 36

4.12 The state diagram of the commit operation. . . . . . . . . . . . . . . . . . . 37

4.13 The implemented commit message input. . . . . . . . . . . . . . . . . . . . . . 38
4.14 The implemented commit feedback dialog. . . . . . . . . . . . . . . . . . . . . 39
4.15 The GitHub repository content. . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.16 The Blame result for the “Bookings” web screen. . . . . . . . . . . . . . . . . 41
4.17 The output of the used blame command. . . . . . . . . . . . . . . . . . . . . . 43
4.18 The implemented blame screen. . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.19 The Fetch operation feedback. . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.20 The History result, which shows the full commit history of the repository. . . 45

5.1 An example of an Action content. These four elements that constitute this action are called action nodes. . . . . . . . . . . . . . . . . . . . . . . . . . . 55

C.1 Participants answers to question one of the post-test questionnaire. . . . . . . 73
C.2 Participants answers to question two of the post-test questionnaire. . . . . . 74
C.3 Participants answers to question three of the post-test questionnaire. . . . . . 74
C.4 Participants answers to question four of the post-test questionnaire. . . . . . 74
C.5 Participants answers to question five of the post-test questionnaire. . . . . . 75
C.6 Participants answers to question six of the post-test questionnaire. . . . . . . 75
C.7 Participants answers to question seven of the post-test questionnaire. . . . . 76

D.1 State diagram of the source control workflow, while UE4 using Perforce. . . . 77
D.2 State diagram of the source control workflow in Unity, while using Git. . . . 78

# List of Tables

3.1 Availability of the most used features of the studied model and text-based SCS. 20

3.2 An extra set relevant features and information regarding the studied model and text-based SCS. * - Perforce's only released *P4CLI* and *P4Web* tools [41] as open source.
**?** - We could not find enough information, to be able to either confirm or deny the existence of the specified data. . . . . . . . . . . . . . . . . . . . . . . . . . 21

5.1 Demographic information of the ten participants. . . . . . . . . . . . . . . . . . 48

5.2 The mean of the answers the first seven questions of the "About our Tool" section of the post-test questionnaire (Appendix A). . . . . . . . . . . . . . . . 50

5.3 Participants' answers to question eight of the post-test questionnaire. . . . . 50

5.4 Participants' answers to question nine of the post-test questionnaire. . . . . . 51

5.5 Participants' answers to question ten of the post-test questionnaire. . . . . . 52

5.6 The mean of the answers for each of the System Usability Scale (SUS) questions. 53

5.7 The various SUS scores with their acceptability and ratings [4] . . . . . . . . 53

# Listings

4.1 The algorithm that fetches the metadata information . . . . . . . . . . . . 28
4.2 Settings Screen User Stories. . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3 Commit Operation User Stories. . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.4 Blame Operation User Stories. . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.5 Fetch Operation User Stories. . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.6 History Operation User Stories. . . . . . . . . . . . . . . . . . . . . . . . . . 45

# Acronyms

**AMOR** Adaptable Model Versioning.

**API** Application Programming Interface.

**CDO** Connected Data Objects.

**CLI** Command-Line Interface.

**CSS** Cascading Style Sheets.

**CVS** Concurrent Versions System.

**DSL** Domain Specific Language.

**EMF** Eclipse Modeling Framework.

**IDE** Integrated Development Environment.

**OML** OutSystems Modeling Language.

**SCCS** Source Code Control System.

**SCM** Source Control Management.

**SCS** Source Control Systems.

**SQL** Structured Query Language.

**SUS** System Usability Scale.

**SVN** Subversion.

**UE4** Unreal Engine 4.

**UI** User Interface.

**UML** Unified Modeling Language.

**VCS** Version Control Systems.

**VPL** Visual Programming Language.

**XML** eXtensible Markup Language.

Chapter 1

# Introduction

Since development teams started to grow in size, the usage of Source Control Systems (SCS) is highly required in order for developers to organize, collaborate and track changes in their projects throughout their development process. Most importantly, SCS give a vision over *what* was changed in a project, *why* it was changed, and *who* changed it. This allows for a controlled and efficient management of the life cycle of a project [27].

There are some alternative terms that can be used to refer to SCS, naming: Version Control Systems (VCS), Source Control Management (SCM) and Source Code Control System (SCCS). SCM is the same as VCS. SCCS was the main *"source code control on Unix platforms for many years"* [20], but the term can still be used to refer to SCS. For the sake of simplicity and coherence, the term that will be used from now on will be SCS.

Although having many advantages, SCS are mainly text-based tools, i.e., mainly built to manage text-based files (source code). This raises a problem when these systems need to handle visual artifacts, usually in the form of binary files: they start to lose their source control functionalities, such as changelogs, file comparisons, blames, among others. As these files are approached as black boxed atomic parts, any change that is done to them will not be well interpreted by regular source control operations, because most of these files require a visual representation of the modifications done to them instead of the plain binary text visualisation, which most of the times, is not readable by the user.

This problematic of SCS handling visual artifacts, emerged with the rise of VPL in the software development industry. VPL are often associated with low-code platforms, which have also been growing in popularity in the past years, as software development teams are in constant pressure to quickly deliver a better, more efficient and reliable software.

As Nigel Warren states [31]:

> *"Web and mobile development demand is booming, challenging IT departments to keep up despite limited resources. As a result, IT is on the hunt for app development solutions that can overcome this challenge. Hence the rapidly increasing adoption of low-code application development platforms over the last two years."*

This leads us to the foundation of this thesis which is to integrate VPL with SCS in the software development process - in this case, to integrate a SCS with the OutSystems platform - as OutSystems is looking for new ways to improve collaboration between developers that use the platform. To achieve this, we intend to leverage on the current power of SCS by bringing them to the OutSystems environment, thus re-utilizing the already provided operations by these SCS. Furthermore, software development in OutSystems is done recurring to the OutSystems VPL, thus raising the same problems described before, about SCS management of visual artifacts, which is widely common in software produced with VPLs.

## 1.1 Motivation

The current version control scenario provided by OutSystems is shown in figure 1.1.

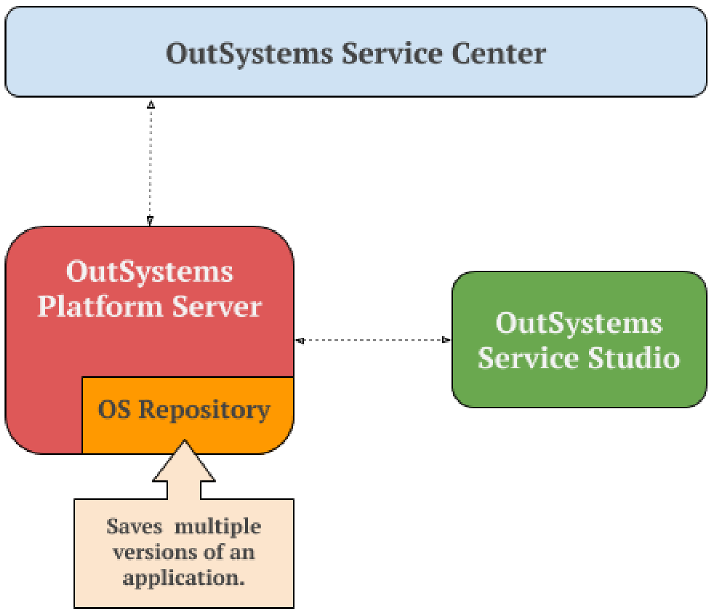

OutSystems Service Center
OutSystems Platform Server
OS Repository
OutSystems Service Studio
Saves multiple versions of an application.

Figure 1.1: Schema of the current state of version control in the OutSystems platform.

As we can obverse, OutSystems is already able to store versions of a project, in the *OS Repository* that resides in the OutSystems Platform Server. Furthermore, OutSystems already provides a few source control operations that the developers can use during the course of their project's development, namely, version history, version tagging, merging and compare facilities. Despite of the project's versions storing, OutSystems does not have the concept of commit. Only the compiled and executable versions are stored, which prevents storing versions that do not compile or execute yet. Moreover these operations were developed by OutSystems from scratch, in order to suit its needs when managing the OutSystems application files, which are binary files.

One downside of the current OutSystems approach, is the fact that the number of operations is rather limited, and here is where OutSystems intends to innovate and bring more value to their product. An integration with an existing SCS would allow OutSystems to harness the SCS operations' power, thus increasing the number of the operations available in the platform. This would also increase the possibilities of collaboration between OutSystems developers, as they would have a more complete set of operations to use in their software development workflows.

Many of these developers are already familiarized with the world of source control and its benefits in projects management, thus started to advise OutSystems to incorporate the required source control functionalities, in order to suit their needs. For instance, Carlos Ardiles, an OutSystems developer, says in a forum thread that [57]:

> *"(...) A blame tool would be useful for specific Client/Server Actions.*
>
> *To be able to right click on an action and click on a blame option. A pop up like the "open another version" window could appear in the screen with the users and the changes made to that specific element."*

This feedback message sent by the OutSystems community, reveals the space for improvement in the area of SCS for the platform. In addition, the fact that this user suggests the incorporation of one operation - the blame operation - that is already provided by most of the SCS, reveals that the best approach to tackle these requests is to complement the platform by integrating it with an existing SCS.

## 1.2 Proposed Solution

Our goal is to enhance the OutSystems platform with an existing SCS, while addressing the associated problems when integrating text-based SCS with VPL: the OutSystems language. To do this, we must develop a solution that prevents the chosen SCS from losing its functionalities while handling visual artifacts produced by the OutSystems platform. Therefore, we propose a system that will act as a *man-in-the-middle* between the OutSystems Service Studio, the developing environment, and the chosen SCS, like it is shown in figure 1.2.

In contrast to figure 1.1, figure 1.2 shows the new addition to OutSystems platform. The source control operations will be called from the Service Studio, where they will be forwarded to our *System*. The *System's* job is to process these operation requests,

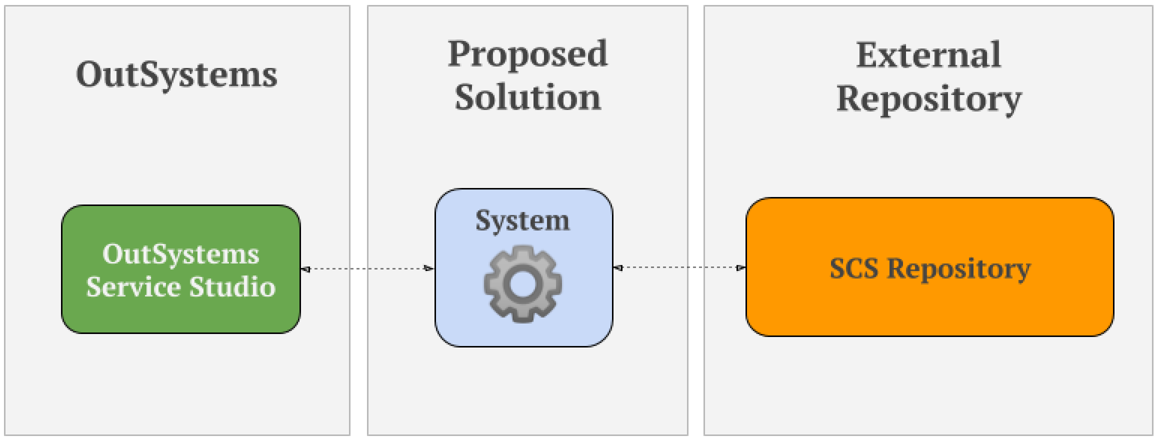


Figure 1.2: Schema of the proposed solution that establishes the communication between the OutSystems Service Studio, the chosen SCS and its external repository.

originated in the Service Studio, and execute the associated source control operations of the chosen SCS. Some of these operations, that are requested by the user from the Service Studio, can be directly translated to the native SCS operations. For example the commit or push operations, because these operations only transfer data. However, operations that modify data or require visual constructs to be displayed, such as, solving merge conflicts or applying blames, are the kind of operations that need to be addressed differently and cannot be directly translated into native source control operations. To tackle this problem, a textual representation of the visual constructs will be used, so that the SCS can function with these constructs. Also, the communications with the external repository will be handled by our system. This means that any information that is sent or received from the Service Studio and from external repository, will be interpreted and processed by our system and forwarded accordingly to their respective recipients.

## 1.3 Key Contributions

With the conclusion of this thesis, there is a state of the art specification of SCS - described in this document - with their extensibility and integration capabilities. We produced a proof of concept for the configuration and integration of GitHub with OutSystems, namely the OutSystems Service Studio. This proof of concept is a system that provides some SCS operations and a connection to an external repository through an intuitive UI. Lastly, this work will contribute to the scientific community as a case study of a SCS integration with a visual software development tool: OutSystems. Because OutSystems is in the domain of VPLs, integrating a text-oriented SCS - Git - proves to be an interesting challenge if we want to preserve its functionalities while correctly handling visual artifacts.

## 1.4 Structure

The remaining chapters of this document are organized as follows:

- Chapter 2 - SCS and Low-Code Software Development: in this chapter key concepts about SCS and the low-code software development are discussed, followed by background information about OutSystems.
- Chapter 3 - The Different Shapes of SCS: this chapter showcases the state of the art of some SCS, with an emphasis on their integration capabilities with external systems, operations provided and ability to manage visual artifacts.
- Chapter 4 - Integrating Git with OutSystems: in this chapter a detailed explanation of the implementation and architecture of the developed proof of concept is described.
- Chapter 5 - Evaluation: this chapter describes the evaluation process of the developed system by showcasing the user tests performed and discussing the obtained results.
- Chapter 6 - Conclusions: in this chapter a discussion of the work developed throughout the course of this thesis, is presented. Also a few insights about the future work are discussed.

# Chapter 2

# Low-Code Software Development

In this chapter, we will introduce the low-code software development concept. An overview of OutSystems as a company and as a product, will also be presented.

## 2.1 Low-Code Software Development

Low-code is the design and development process of a software system by using minimal hand-written source code [45]. Low-code development techniques usually recur to visual components as a way to mask and automatically generate source code, thus allowing developers to quickly build a piece of software. As Matthew Revell says [45]:

> *"Developers skip all the infrastructure and re-implementation of patterns that can bog them down and go straight to the unique 10% of an application."*

i.e., developers start working on the specific part of their software that actually represents the value and the uniqueness of the product they are building. Furthermore, the low-code approach allows people that are not familiarized with writing source code, to actually partake on a software development project, as the hand-writing source code is kept to a minimum.

## 2.2 OutSystems

OutSystems[1] is a software company that is primarily focused on improving developers efficiency whilst developing software on their daily basis. OutSystems is capable of achieving this by providing developers with methodologies that accelerate, abstract and ease the majority of software development tasks.

[1]The OutSystems website - https://www.outsystems.com/

OutSystems' product is a low-code platform that accelerates the software development process. The OutSystems platform allows developers to build, deploy and manage, platform-independent mobile and web applications. By being in a low-code environment, building an application becomes a fast and productive process [1].

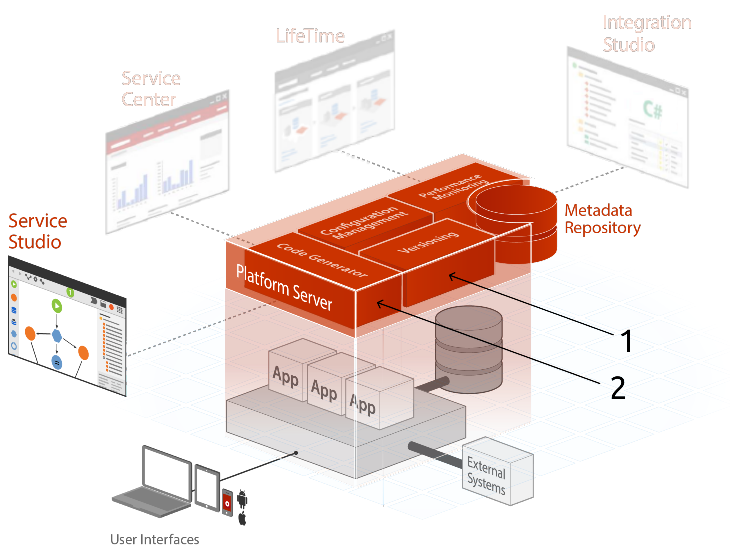


Figure 2.1: The OutSystems Platform components, with an emphasis on the Service Studio and the Platform Server [38].

Figure 2.1 shows the two OutSystems platform components that are going to be approached next: the Service Studio and the Platform Server.

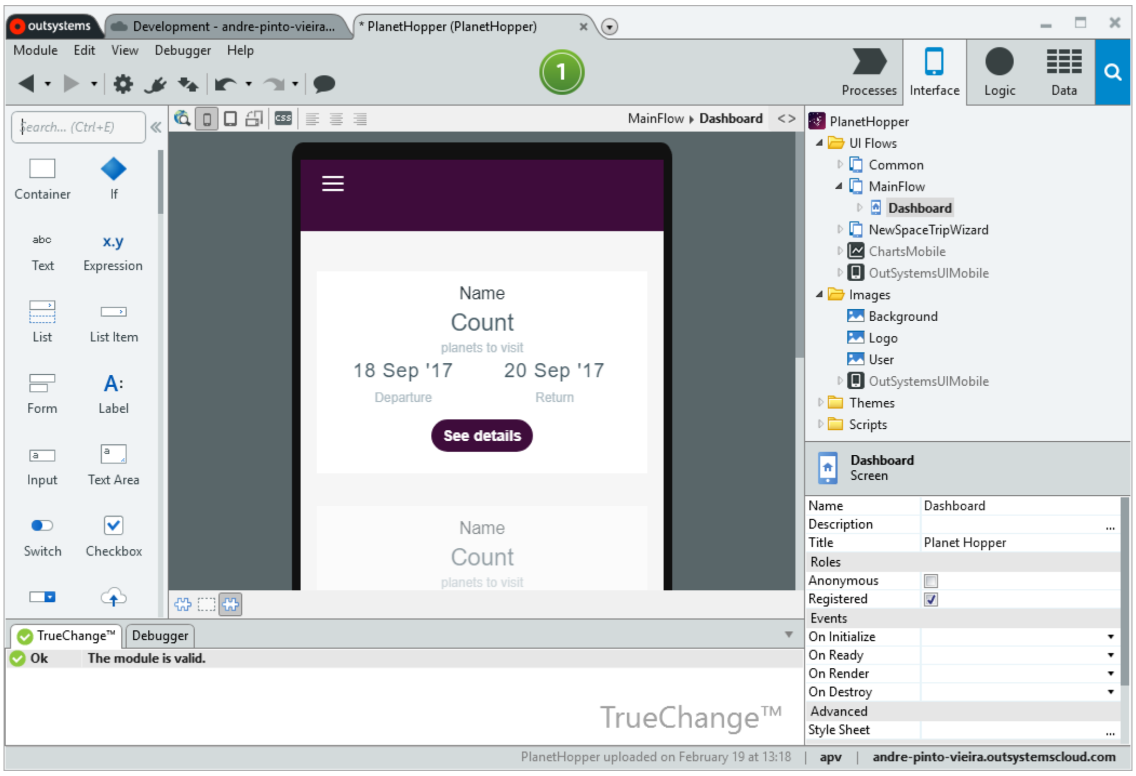
### 2.2.1 Service Studio

The Service Studio is the OutSystems visual development environment [38]. It is where the developers build applications in a *drag-and-drop* style. With the Service Studio, developers can create:

**The application's UI** Built through a composition of visual widgets, like buttons or text boxes, like it is represented in figure 2.2.

**The application's logic** Built through the composition of visual widgets, like variable assigns or conditional and loop primitives. This composition is aggregated in a graph-like structure that represents the step-by-step operation workflow, like it is shown in figure 2.3.

**The application's database modeling** Where the applications' databases are modeled through the creation of entities and static entities, like it is displayed in figure 2.4.

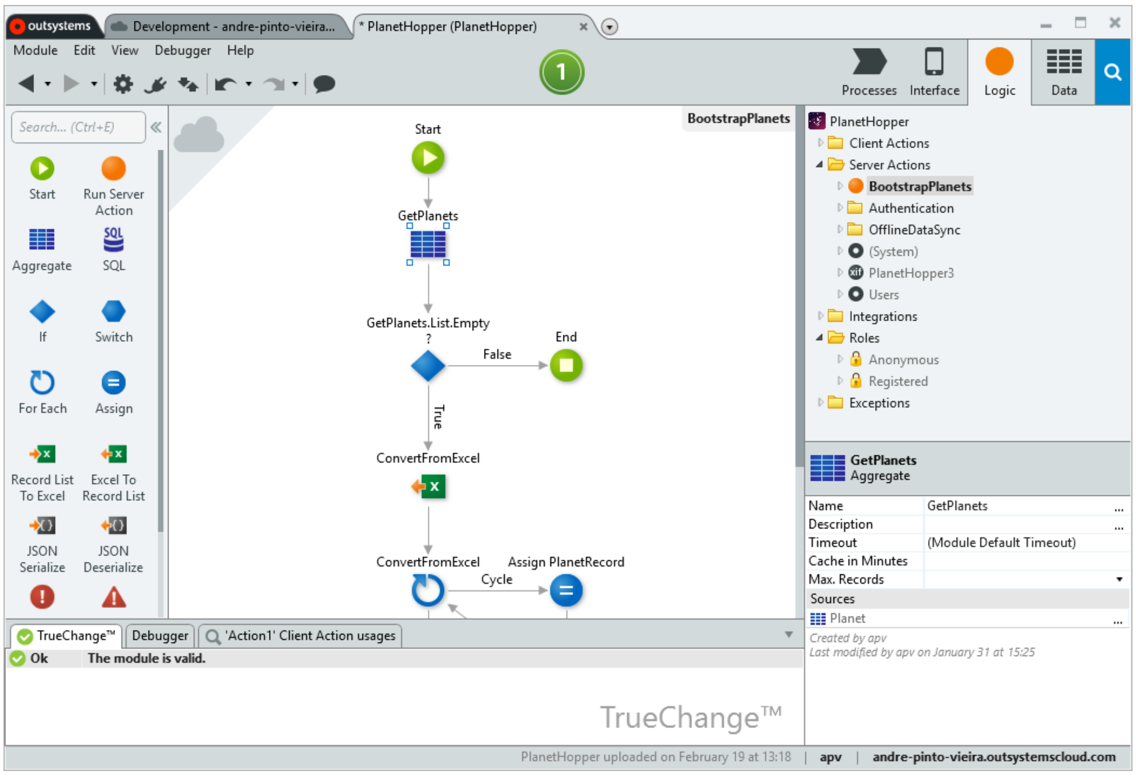


| | |
|---|---|
| Name | Dashboard |
| Description | ... |
| Title | Planet Hopper |
| Roles | |
| Anonymous | ☐ |
| Registered | ✓ |
| Events | |
| On Initialize | |
| On Ready | |
| On Render | |
| On Destroy | |
| Advanced | |
| Style Sheet | ... |

Ok The module is valid.

PlanetHopper uploaded on February 19 at 13:18 | apv | andre-pinto-vieira.outsystemscloud.com

Figure 2.2: Service Studio overview while editing an application's UI.



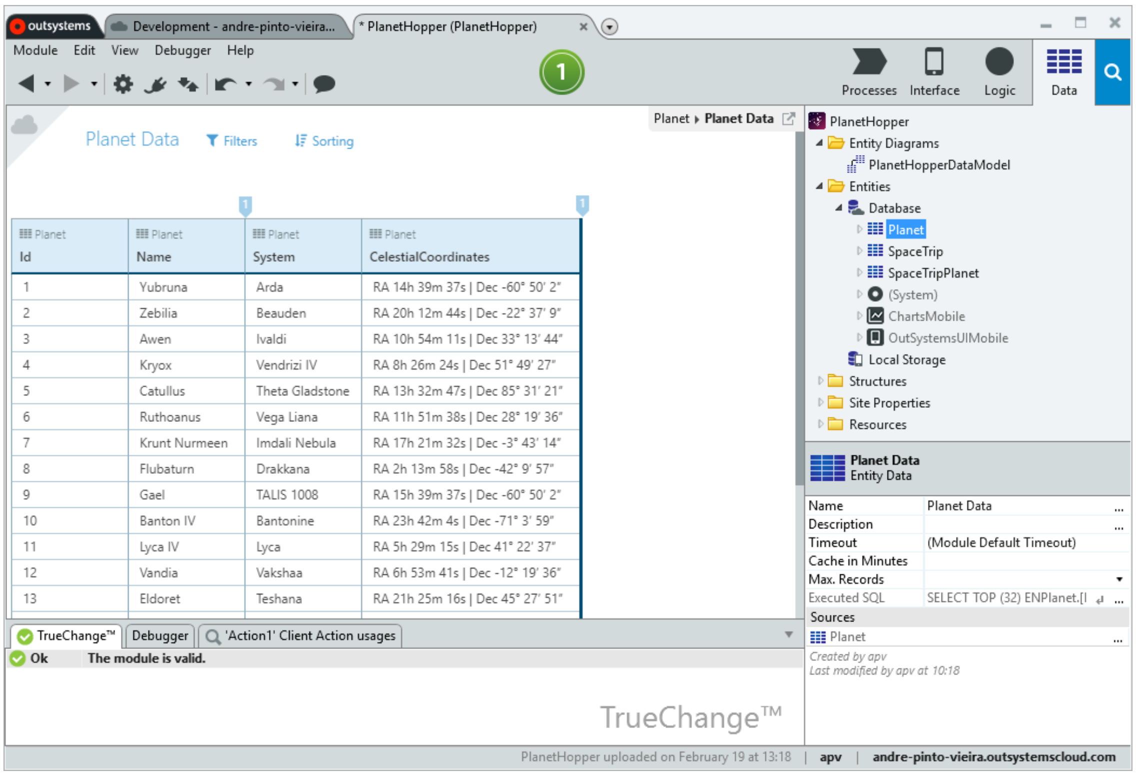
| | |
|---|---|
| Name | GetPlanets |
| Description | |
| Timeout | (Module Default Timeout) |
| Cache in Minutes | |
| Max. Records | |
| Sources | |
| Planet | ... |

Created by apv
Last modified by apv on January 31 at 15:25

Ok The module is valid.

PlanetHopper uploaded on February 19 at 13:18 | apv | andre-pinto-vieira.outsystemscloud.com

Figure 2.3: Service Studio overview while editing an application's logic.



| Planet Id | Planet Name | Planet System | Planet CelestialCoordinates |
|---|---|---|---|
| 1 | Yubruna | Arda | RA 14h 39m 37s \| Dec -60° 50′ 2″ |
| 2 | Zebilia | Beauden | RA 20h 12m 44s \| Dec -22° 37′ 9″ |
| 3 | Awen | Ivaldi | RA 10h 54m 11s \| Dec 33° 13′ 44″ |
| 4 | Kryox | Vendrizi IV | RA 8h 26m 24s \| Dec 51° 49′ 27″ |
| 5 | Catullus | Theta Gladstone | RA 13h 32m 47s \| Dec 85° 31′ 21″ |
| 6 | Ruthoanus | Vega Liana | RA 11h 51m 38s \| Dec 28° 19′ 36″ |
| 7 | Krunt Nurmeen | Imdali Nebula | RA 17h 21m 32s \| Dec -3° 43′ 14″ |
| 8 | Flubaturn | Drakkana | RA 2h 13m 58s \| Dec -42° 9′ 57″ |
| 9 | Gael | TALIS 1008 | RA 15h 39m 37s \| Dec -60° 50′ 2″ |
| 10 | Banton IV | Bantonine | RA 23h 42m 4s \| Dec -71° 3′ 59″ |
| 11 | Lyca IV | Lyca | RA 5h 29m 15s \| Dec 41° 22′ 37″ |
| 12 | Vandia | Vakshaa | RA 6h 53m 41s \| Dec -12° 19′ 36″ |
| 13 | Eldoret | Teshana | RA 21h 25m 16s \| Dec 45° 27′ 51″ |

Ok The module is valid.

| | |
|---|---|
| Name | Planet Data |
| Description | |
| Timeout | (Module Default Timeout) |
| Cache in Minutes | |
| Max. Records | |
| Executed SQL | SELECT TOP (32) ENPlanet.[I |

*Created by apv*
*Last modified by apv at 10:18*

PlanetHopper uploaded on February 19 at 13:18 | apv | andre-pinto-vieira.outsystemscloud.com

Figure 2.4: Service Studio overview while editing an application's database.

#### 2.2.1.1 Platform Server

The Platform Server is the core of the OutSystems platform. It is responsible for generating, optimizing, compiling and deploying the applications built in the Service Studio [38].

To deploy an application, the Service Studio sends the applications' models to the Platform Server, where as figure 2.1 illustrates the applications are:

- Versioned, by using the OutSystems versioning system - identified by the number *1 - Versioning*, in figure 2.1 - which will be approached in detail in section 3.2.1;
- Compiled into .NET and optimized for performance and security reasons - identified by the number *2* - *Code Generator*, in figure 2.1;
- Deployed to the front-end OutSystems servers, which establish the contact between the user and the application.

### 2.2.2 The OutSystems Application Files

All of the applications that are built in the Service Studio, are transported and stored as OutSystems Modeling Language (OML) files. The OMLs are binary files that have an underlying eXtensible Markup Language (XML) specification [30]. This XML specification can later be extracted to a file with the aid of an internal OutSystems tool, that decodes the OMLs to XML.

Chapter 3

# The Different Shapes of SCS

In this section, we will introduce Source Control Systems (SCS) and approach source control methodologies in software systems, with a variety of SCS being presented.

## 3.1 Source Control Systems

The need to organize and track changes in a project exists for as long as people started thinking about collaboration and the concept of *versioning*: to store and manage multiple copies of a project, since its creation until its completion [50]. The possibility of tracking and review changes or the possibility to restore a previous version, are also motivations that lead to the creation of SCS [27]. These systems are intended to provide a way for developers to work concurrently on the same project, while helping them in the endeavour of managing the project's source code. Nowadays, in the software development industry, the concept of source control has become associated with ways to track code changes, allow the revision of the code history, provide the ability to revert to previous versions, provide merge and comparison facilities of distinct versions of a project and to allow the collaboration of multiple individuals on the same project [27, 59]. Even though different SCS may provide complementary operations, they all try to respect and follow these basic guidelines as to what a SCS should be.

## 3.2 Source Control in Low-Code Platforms

Besides the OutSystems appearance in the low-code development world, innovations in the low-code area are rising, thus another promising competitor emerges: Mendix. The Mendix use-case is the closest for the study of source control in low-code platforms besides OutSystems, and will both be introduced next.

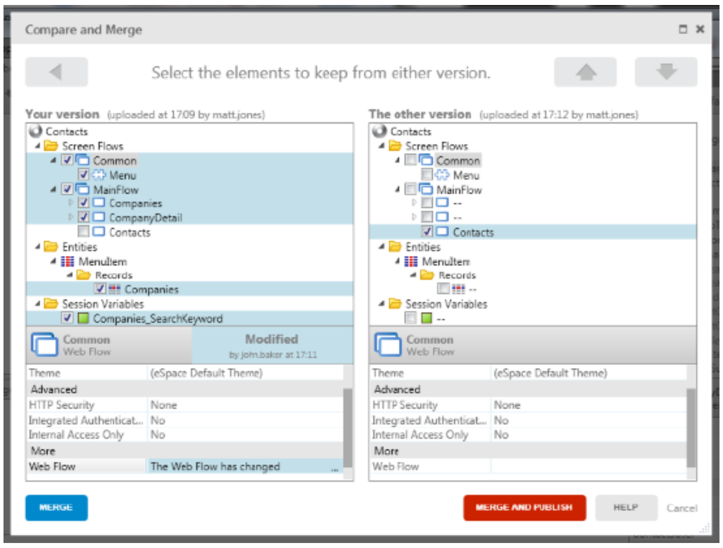
### 3.2.1 Versioning in OutSystems

OutSystems currently supports version control through an internally developed versioning system that is able to provide some version control operations namely, comparing and merging, version history, version tagging, rollbacks to previous versions, and developer sandboxes [26, 53]. Moreover, OutSystems does not integrate with any existing SCS that already partake in the software development of its users. This is a limiting factor when it comes to the source control operations available in the current versions of the OutSystems platform, and because users cannot use the same source control methodologies when working with OutSystems..

Lastly, OutSystems keeps the versions of the applications in a central repository that resides in the developer's *personal environment*[1]. Each time an application is published, i.e. when the developer uploads the application to his personal environment, a new version is stored and made available for everyone with access to it.

#### Compare and Merge

When a developer wants to merge two versions of an application, the OutSystems Service Studio will automatically perform the merge or, if the developer requests or there were any conflicts detected, it allows to manually perform the merge. The OutSystems Service Studio provides three ways to compare and see differences of two versions of an application: in a widget tree view, like it is observed in figure 3.1, in a text view, like it is displayed in figure 3.2 and in a graph-like view, like it is shown in figure 3.3.



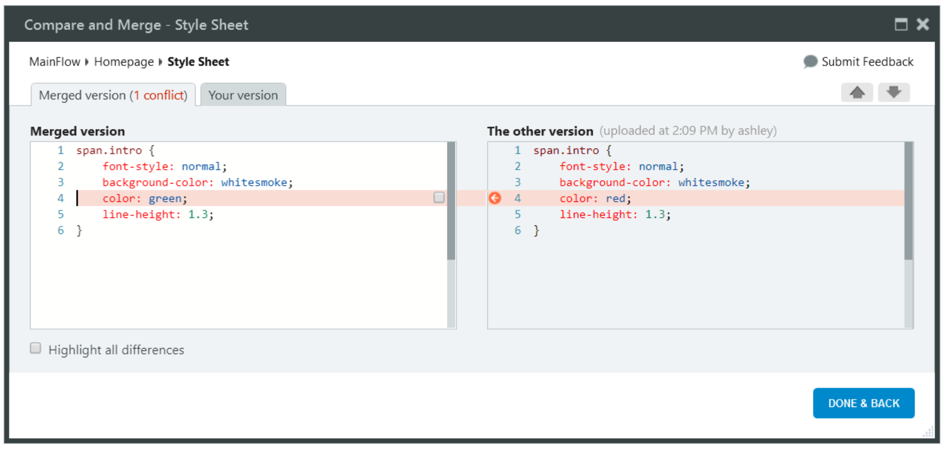
| | |
|---|---|
| Theme | (eSpace Default Theme) |
| Advanced | |
| HTTP Security | None |
| Integrated Authenticat... | No |
| Internal Access Only | No |
| More | |
| Web Flow | The Web Flow has changed ... |

| | |
|---|---|
| Theme | (eSpace Default Theme) |
| Advanced | |
| HTTP Security | None |
| Integrated Authenticat... | No |
| Internal Access Only | No |
| More | |
| Web Flow | |

Figure 3.1: Merge situation, where the developer can visually compare two versions and merge them manually [26].

[1]The personal environment is the cloud-based version of the OutSystems platform. For more information refer to `https://success.outsystems.com/Support/Personal_Environment/What's_an_OutSystems_personal_environment%3F`.

The widget tree view is the default view for a merging situation. Like we can see in figure 3.1, the changed application widgets appear highlighted and displayed in a tree.


Compare and Merge - Style Sheet
MainFlow ▸ Homepage ▸ Style Sheet
Submit Feedback
Merged version (1 conflict)
Your version
Highlight all differences
DONE & BACK

Merged version

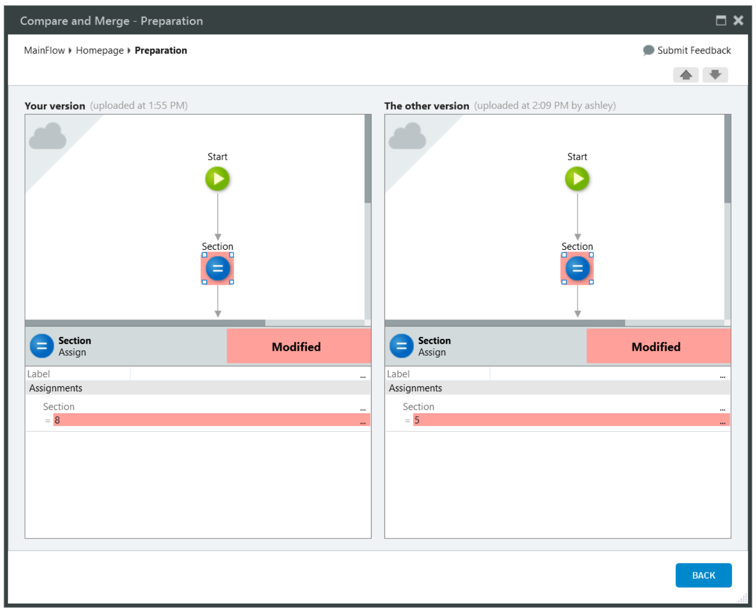
```
span.intro {
    font-style: normal;
    background-color: whitesmoke;
    color: green;
    line-height: 1.3;
}
```

The other version (uploaded at 2:09 PM by ashley)

```
span.intro {
    font-style: normal;
    background-color: whitesmoke;
    color: red;
    line-height: 1.3;
}
```

Figure 3.2: Merge conflict situation between two versions of a CSS file [9].

The text view is really simple and it is widely used as the mean of showing differences in most of the SCS. It is where changes are seen through the comparison of lines of text, like we can see in figure 3.2. This view is only shown for the comparison of text artifacts that are used when programming with OutSystems, like Structured Query Language (SQL) statements, Cascading Style Sheets (CSS) and JavaScript files.


Compare and Merge - Preparation
MainFlow ▸ Homepage ▸ Preparation
Submit Feedback
Your version (uploaded at 1:55 PM)
The other version (uploaded at 2:09 PM by ashley)
Start
Section
Section
Assign
Modified
Label
Assignments
BACK

Your version: Section = 8

The other version: Section = 5

Figure 3.3: Merge conflict situation between two versions of an action [9].

Lastly, the graph-like view is shown when the developer needs to visualize differences between two versions of an *action*[2]. Figure 3.3 shows a merge conflict between two versions of an action in the value assigned to the variable *section*.

[2]Actions are a part of the logic elements of the OutSystems language, allowing developers to describe the intended behaviour of their application.

#### Version Tagging

Version tagging [26] in OutSystems is intended to facilitate developers with the ability to tag a specific version of the application, as a *stable* state of the application was obtained or a set of objectives were fulfilled. By tagging the versions of the application during its development process, developers can easily deploy it to production when a certain goal is achieved or, if needed, return to a previous state that was stable.

#### Version History and Rollbacks

In OutSystems, as every publish or version of an application is stored with the information of the date and author of the publication, it is possible to retrieve and rollback to any of these previously saved versions.

#### Developer Sandbox

Developer sandboxes [26] are private areas of an OutSystems application, where developers can concurrently work on the same application without interfering with each others work, by publishing their versions to this private area. One can think of developer sandboxes as a branch of the public version of the application. When their work is concluded, then they can publish and merge it with the public version of the application.

### 3.2.2 Versioning in Mendix

Mendix is a low-code platform that provides a fast and easy development, management and deploy processes for web and mobile applications [33, 49]. These applications are also built through a drag-and-drop style like we saw with OutSystems. Mendix adopted the centralized version control workflow, as its SCS is built on top of Subversion (SVN), which is a centralized SCS. Mendix integrates with SVN as a way to benefit from the existing operations provided by this system, while also granting a familiar workflow to the users that already worked with SVN [54].

Mendix also provides its own SVN repository hosting service, called Mendix Team Server. However, some companies policies require their software to be stored in servers on their premises. To tackle this issue, Mendix also provides a way for users to configure their own repositories, thus allowing them to oblige to their company policies [56].

Despite the fact that Mendix is able to directly harness most of SVN's operations, SVN's merge operation cannot be used in the model files, which are the Mendix's large binary files. This lead Mendix to create an internal algorithm, that is responsible for automatically merge changes or, if conflicts emerge, allow the users to manually merge them [54]. This custom merge algorithm consequently lead to the creation of a differentiation algorithm as well, which calculates differences between model files and feeds them to the merge algorithm, while being the backbone structure for showing differences when the user is manually performing a merge of model files [56].

Finally, the Mendix use case revealed to be an interesting case study as it enables a text-based source control (SVN) integration with a visual programming language (Mendix).

## 3.3 Source Control in IDEs

Integrated Development Environments (IDEs) provide facilities for developers to build software, and since its coexistence with SCS, the latter revealed to be a great complement to the IDEs range of functionalities. The integration of SCS with these environments, eases the software development process, as developers can follow the source control workflows from within the IDE they are working with.

There are a lot of examples regarding SCS integration with IDEs, for example, the integration of Git, Mercurial or Perforce with Visual Studio or even SVN, Git or Mercurial integration with Eclipse. We chose the Unity and Unreal Engine 4 (UE4) IDEs use cases because the matter of integrating a SCS with these two systems, are case studies that fall under the topic of a SCS handling visual artifacts, as these two IDEs manage binary files and visual assets.

### 3.3.1 Source Control in Unreal Engine 4

Unreal Engine 4 (UE4)[3] [46] supports by default two SCS: SVN and Perforce. Nevertheless, developers are free to choose which system they want to use, as the engine provides the same built-in interface regardless of the chosen SCS [47]. However, the operations that this interface allows developers to perform, may not be the complete set of operations that are provided by the SCS itself, i.e., there may be some operations, like the visualization of branches in a tree, that are not displayed in the UE4 IDE. Nonetheless, such operations can still be concurrently used with the respective SCS client or via the Command-Line Interface (CLI).

Large binary files are very common to exist in projects developed with UE4, as they are usually used to describe compositions of visual artifacts. Therefore, while using Perforce, UE4 recurs to file locking [52], to lock binaries that are being edited by one user, so that others cannot concurrently change it. This lock is later released when the developer finishes his work by committing his changes.

Figure D.1 illustrates the source control workflow of Perforce. It briefly describes the control and communication flow, between calls of source control operations and the engine's response. To use the source control operations, the user is provided with two choices: to use the integrated operations that are available via UE4 UI, or to use the operations directly via Perforce's client. There is only one setback regarding the usage of the integrated source control operations, and that is the fact that the available operations

[3] UE4 is a set of tools that allows creators to build applications, animations and high-quality games. For more information refer to `https://www.unrealengine.com/en-US/features`.

are limited. So, if one intends to use Perforce to its full potential, one will eventually need to recur to Perforce client.

### 3.3.2 Source Control in Unity

Unity[4] [51] integrates with two version control systems: Perforce and PlasticSCM, by featuring an interface for both of these SCS. This interface is integrated in the Unity IDE and allows for the basic source control operations to be performed: add, commit, push, pull. More complex operations like branching, showing logs or visualizing differences, although they can be called from within Unity, in the case of showing differences and branches in PlasticSCM, they are, however, displayed in PlasticSCM's client, where proper visualization techniques are implemented. The usage of different SCS like Git or Mercurial is also possible in Unity. There are two ways to tackle this: either by using an external SCS client or the CLI, to perform any of the version control operations required; or by using Unity plugins that integrate with the IDE. Furthermore, among others, these plugins are intended to provide a way to integrate SCS like Git, SVN or Mercurial, with the Unity IDE.

Using the *GitHub for Unity* [19] plugin as an example, it enables the integration of Git with the Unity IDE, while using GitHub as the hosting service. The plugin provides the following SCS operations [19]: initialize a repository, fetch, push and pull from the repository, add and commit to the local repository, create and switch branches, showing changes and history, and file locking.

Figure D.2 illustrates the source control workflow of using Git operations via the plugin *Github for Unity*. This workflow is very similar to the one represented in figure D.1, where the user is also able to choose whether or not he/she intends to use the integrated source control operations or remain with the classic CLI or source control client to use them. Once again, other source control operations, like rebase, grep or tagging in Git, can still be used side-by-side with *GitHub for Unity* plugin, although they can only be called via CLI or a specific Git client.

### 3.3.3 Summary

The SCS integration with the UE4 and Unity IDEs demonstrated the possibility of integrating a text-oriented SCS with systems that manage visual artifacts. Unity has a textual representation for all of its files, thus not showing great problems when integrating and using SCS. UE4, however, suggests the usage of file locking to handle modifications in visual artifacts. This prevents conflicts and any merge necessity over these visual files, thus enabling the version control workflow to proceed smoothly. Essentially, UE4 solves the issue of handling visual artifacts, by not causing disruptions among the versioned visual files.

[4]Unity is a technology that allows creators to build games, animations and virtual prototypes. For more information refer to `https://unity3d.com/`.

## 3.4 Source Control in Modeling Languages

Modeling Languages like the Unified Modeling Language (UML) are within the world of VPLs, which make them an interesting case study when SCS are used to manage models that were built with them. Thus, we will discuss some of the existing tools. Furthermore, in addition to UML models, some of these tools provide source control facilities to Eclipse Modeling Framework (EMF)[5] models.

Note that some of the tools that will be presented are not necessarily SCS as a whole. Some of them are just tools that implement one or more operations that are in the world of source control, for instance the compare and merge operations, while others can be *hubs* for other tools.

**EMFStore** It is an integrated tool in Eclipse, designed specifically for EMF-based models. It provides compare, merging, branching, tagging and history facilities, featuring an UI where conflicts, branches and history are displayed in a visual manner. EMFStore recurs to the EMFCompare tool to provide the merge and compare functionalities [15] and has its own repository server architecture, prepared for EMF models, where users share and commit their models [28, 58].

**Connected Data Objects (CDO)** It is an integrated tool in Eclipse, as it is built on top of the EMF. CDO, provides comparison, merge, branching and history functionalities, while supporting a way to change history, that is, to change previous commits. Also, the comparison feature provided by CDO recurs to the EMFCompare tool in order to provide an efficient compare editor. Lastly, CDO provides two manners to work offline: repository cloning, where the user holds an entire copy of the repository - the full commits history and branches - and a lightweight approach, where it is possible for the user to checkout a copy of a specific branch point, so that he/she holds a small chunk of the repository at a given time [7]. This repository is dependent of the EMF [8].

**Adaptable Model Versioning (AMOR)** AMOR provides merging and comparison features by innovating in the following topics: i) in precise conflict detection, by avoiding undetected and wrongly identified conflicts, ii) in intelligent conflict resolution, by providing techniques for the representation of conflicts and suggesting resolution strategies. The comparison feature requires that a previously defined comparison method is provided: the authors recommend the usage of EMFCompare. So AMOR is somewhat dependent on existing software, in order to reach its full potential. In this case, with the usage of EMFCompare, the user is restricted to work on top of EMF models [2].

**ModelBus** It is a tool integration framework, based on a service oriented architecture. It features: the integration of software tools, construction of integrated and automated tool chains and supports collaboration of developers (via VCS). ModelBus already features the integration of SVN and Git as some of its core technologies. It was designed for managing

[5]EMF is a modeling framework and code generation tool that works upon data models EMF. For more information refer to `https://www.eclipse.org/modeling/emf/`.

complex development processes with the need to integrate a variety of tools working together [36].

### 3.4.1 Summary

These systems provide very particular solutions that are grappled to specific problems. For instance, the first three tools - EMFStore, CDO and AMOR - work with EMF models and propose solutions to handle these specific model types, which are structured XMI files. The other tool - ModelBus - provides a tool integration environment. Thus, these tools are not acceptable candidates to perform the integration with the OutSystems platform. Despite OutSystems files being structure as XML files, as described in section 2.2.2, an approach where the solution would work with the underlying XML specification, instead of the raw binary files that the users can see, would present two problems. The first problem is linked to the file structure privacy, that must remain concealed to final users. That is why the application files are obfuscated through a binary representation. The second problem is related to the amount of time the current tool takes to transform a binary file into its XML specification, which is a time-consuming deed. Furthermore, most of the model-based SCS studied are Eclipse plugins, which discards the big necessity for these tools to be standalone software solutions, to be able to integrate with any external systems.

## 3.5 Source Control in Spreadsheets

Regarding source control in spreadsheets, there is an interesting case study that needs to be mentioned: the SheetGit tool [32, 37]. SheetGit provides a Git integration with Excel through an intuitive UI. This tool provides operations such as: compares, merging and the creation of new versions. Also, SheetGit uses BitBucket as the repository host.

SheetGit uses an interesting approach to tackle the problematic of the version control operations loss: the usage of metadata files. These metadata files contain the list of changes done in each commit, the parent commit of the current commit and the current working branch. So, for every commit, the spreadsheet file and the related metadata file containing the said information, are stored in the remote repository for each commit [37]. Thus using an operation-based approach, where the operations made to the spreadsheet files are store in the repository as well.

This work is similar to the proposed solution for this thesis, as discussed in section 1.2, where SheetGit fulfills the gap between the VPLs represented by Excel spreadsheets and a VCS portrayed by Git. Furthermore, the metadata approach appears again in SheetGit, thus revealing to be a promising solution to tackle the version control problem when dealing with visual artifacts.

## 3.6 Source Control in Text-Based Languages

Text-based SCS need to be mentioned because they will be the foundation of the integration we intent to provide to the OutSystems platform. A selection of the widely known text-based SCS will be described, with a description of their main functionalities, operation extension capabilities and integration with external systems.

**Git** Is an open source distributed SCS that is designed to be lightweight with a focus on performance and scalability while handling large sized projects [16]. Furthermore, Git features the widely used operations like: comparing and merging, branching, history revision and version tagging, with a special emphasis on operations such as [16]: File Locking, to ensure that only one user has access to a given file; A Staging area, that acts as a provisional space that holds the modified files before committing, whilst allowing the user to review them; And Stash, that stores the state of a *dirty* working directory (that is not yet committed) back to a clean working directory state that matches the current commit in the repository.

Lastly, Git can be easily integrated with external systems, as we can obverse with the Git integrations done with VSCode [55] or Unity [19], while also providing the facility of adding custom source control operations.

**Bazaar** This SCS [5] has the particularity of allowing for both centralized and distributed workflows to be used. Also Bazaar innovates in the centralized workflow by introducing *bound branches*, that verify if the local repository is up-to-date with the central repository before committing. If the verification is successful, the central commit is performed before the local one, thus ensuring the atomicity of the operation.

Besides the common SCS operations - like merges, compares, branches and history review - it also provides tagging and shelve operations. The shelve operation is the same as Git stash one but with a different name. Lastly, Bazaar also provides the ability to add custom operations via plugins as a way to extend its core functionalities.

**Mercurial** Is a distributed SCS [34] that has the particularity of providing the ability to either modify or extend existing operations and create new ones, in a very intuitive and easy way, via the creation of plugins. Furthermore, besides the common source control operations, Mercurial provides [34]: version tagging, file locking, staging area, like we saw in Git that goes by the name *DirState* and shelve (or stash). Lastly, Mercurial can be integrated with external systems, as VSCode [25] and Eclipse [35] do.

**Concurrent Versions System (CVS)** This SCS deserves an honorable historical mention, as it was one of the precursors of modern era SCS. CVS features operations such as [11, 12]: merging and comparison of source code, branching, history revision and version tagging.

**SVN** Originally designed as an improved CVS, SVN [3] is a centralized SCS built to be reliable and simple to use. SVN provides operations such as [3]: compares and merges, branching, history revision and version tagging, a staging area known as *Changelists* and file locking. Regarding SVN's integration with external systems, the Eclipse [14] and IntelliJ [48] use cases, illustrate the flexibility this tool has, to be complemented with other software.

**Plastic SCM** This SCS also allows for distributed and centralized workflows to be adopted, with an emphasis on the fact that both of these workflows can be used by different users concurrently. Apart from text, Plastic SCM innovates with an image differentiation algorithm that allows users to visually compare two versions of an image. Features that Plastic SCM provides besides the usual source control operations, are [42]: version tagging (named as version labels), *changelists* (also know as staging area), file locking (named as exclusive checkout), and shelve. All of these operations are performed and displayed in a simple and intuitive UI. Plastic SCM's integration with external systems is restricted as existing integrations, like the Unity [43] and Microsoft Office[6] [44] cases, are created when new versions of Plastic SCM are launched.

**Perforce** HelixCore is the the Perforce's SCS software [40], often designated as just Perforce, which leverages from an intuitive and user-friendly UI. Perforce provides source control operations like: merging, branching, history, version labels, also known as tagging, *changelists* or staging area, file locking and lastly, shelve [39]. Perforce is able to integrate with almost any system due to its own plugins and open API [24], as we can see in the examples of the integrations done with Visual Studio [23] and Unity [22]. Also, it is possible to extend operations' functionalities through *triggers*, which are custom programs that are called when specific source control operations are used [21].

Tables 3.1 and 3.2 are a compacted way to visualize and compare, the studied text-based SCS. These SCS revealed to be the ones of utmost interest, because they are flexible and offer the largest number of functionalities.

| | Source Control Features | | | | | | |
|---|---|---|---|---|---|---|---|
| SCS | Compare and Merge | Branches | History | Tagging | Blame | Staging Area | Stash |
| Git | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Bazaar | Yes | Yes | Yes | Yes | Yes | No | Yes |
| Mercurial | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| SVN | Yes | Yes | Yes | Yes | Yes | Yes | No |
| CVS | Yes | Yes | Yes | Yes | Yes | No | No |
| Plastic SCM | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Perforce | Yes | Yes | Yes | Yes | Yes | Yes | Yes |

Table 3.1: Availability of the most used features of the studied model and text-based SCS.

[6]Only in the Microsoft Office Word, Excel and Powerpoint.

| SCS | Open Source | Extensible | Integration with external systems | Latest Version Release |
|---|---|---|---|---|
| Git | Yes | Yes | Open | 2018-12-15 |
| Bazaar | Yes | Yes | Open | 2016-01-31 |
| Mercurial | Yes | Yes | Open | 2018-01-07 |
| SVN | Yes | ? | Open | 2018-10-30 |
| CVS | Yes | ? | Open | 2009-09-03 |
| Plastic SCM | No | ? | Open | 2019-01-11 |
| Perforce | Yes* | Yes | Open | ? |

Table 3.2: An extra set relevant features and information regarding the studied model and text-based SCS. * - Perforce's only released *P4CLI* and *P4Web* tools [41] as open source. **?** - We could not find enough information, to be able to either confirm or deny the existence of the specified data.

## 3.7 Choosing the SCS to integrate with OutSystems

Table 3.1 features a set of known SCS operations. These operations were chosen from a list of SCS operations that were internally discussed at OutSystems, as operations of interest. As the table shows, the represented SCS are able to provide the essential and most common source control operations: compare, merge, branching, history revision and version tagging. However, it is in the last three functionalities that a noticeable discrepancy rises between them. Git, Mercurial, Plastic SCM and Perforce are able to hold their positions as the SCS that are able to provide all of the considered functionalities, whereas Bazaar, SVN and CVS cannot provide the same functionalities as the other SCS. To elect a tool between Git, Mercurial, Perforce and Plastic SCM, it is necessary to gather additional information about these systems. As it is displayed in table 3.2, information regarding the extensibility of SCS operations, the possibility of SCS integration with external systems, the latest released version and the availability of the SCS source code to the public, is described.

By taking into consideration this new information, our goal is to find a SCS that can easily be integrated with external systems and provide an easy way to add or extend its functionalities. As the software development industry is in constant growth, it is also important to favor a SCS that is regularly updated. Also, a software that is Open Source is preferred as it can be easily edited to fit any project's needs, while relying on a large number of people that inspect the code base looking for possible bugs. Considering these last four factors, that are represented in table 3.2, we can exclude Plastic SCM and Perforce. Moreover, these systems operations extension capabilities are not very easy and fast to use. That leaves us with Git and Mercurial - that are very much alike - which will be compared in the following section.

### 3.7.1 Mercurial Vs. Git

Mercurial and Git are both powerful SCS, with several similarities. Both are able to provide operation extension facilities, which allow the creation of new operations or the extension of the existing ones - like the diff and merge operations, where one can change the default algorithm - as one sees fit. It is noticeable that Mercurial's syntax is simpler than Git's, thus revealing to be a weak point of Git, especially because new Git users may have difficulties while learning and using it. Such simplicity also comes with the cost of having fewer available operations when compared to Git, which provides a wider range of source control operations. Moreover, Git's popularity has risen, and became the most used SCS with much bigger community than Mercurial's, as we can see in figure 3.4. Thus, we choose Git.

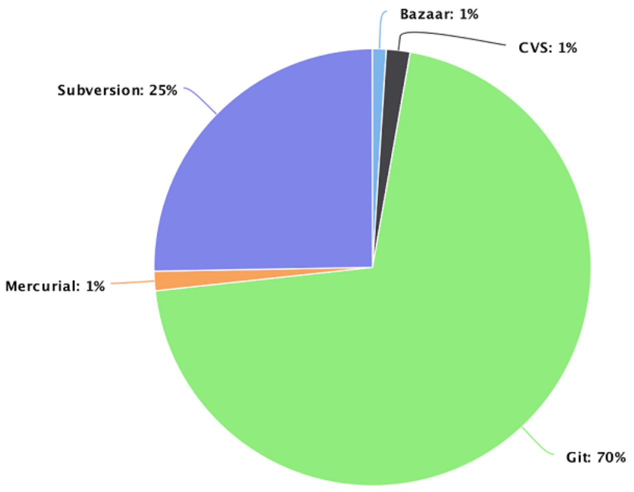


Figure 3.4: Number of repositories of each SCS as of July 2019 [10].

### 3.7.2 The Chosen Git Operations

According to the operations described in table 3.1 in section 3.6, we intend to provide: compare and merging, history and blame. Moreover, the blame operation was highly requested by some employees when we discussed our solution with OutSystems. The commit and fetch operations also need to be provided, despite not being mentioned in the table, because these two operations are elementary SCS operations that transfer data and are present in every SCS.

The decision to include these five operations resides in the fact that these operations are enough to provide an intermediate SCS usage workflow. This means that the user will be allowed to perform the basics of source control, which consists of doing commits, fetch changes, or see history. Also, the user will still be able to perform more complicated operations, such as compare and merges to solve possible conflicts or applying blames. The branching operation could be interesting to incorporate, however it was discarded because it created a new problem that is outside the scope of this thesis, and that is related to the database architecture of the OutSystems applications.

CHAPTER 4

# Integrating Git with OutSystems

In this chapter, a description of the system development and architecture will be discussed, focusing on design decisions, solutions to the problems found and operations descriptions.

## 4.1 Overview

As discussed in section 1.2, we proposed a system that acts as a *man-in-the-middle* between the OutSystems Service Studio and an external repository. As the chosen SCS is Git, we selected GitHub as the hosting service for the Git repositories. We term our system OSGit.

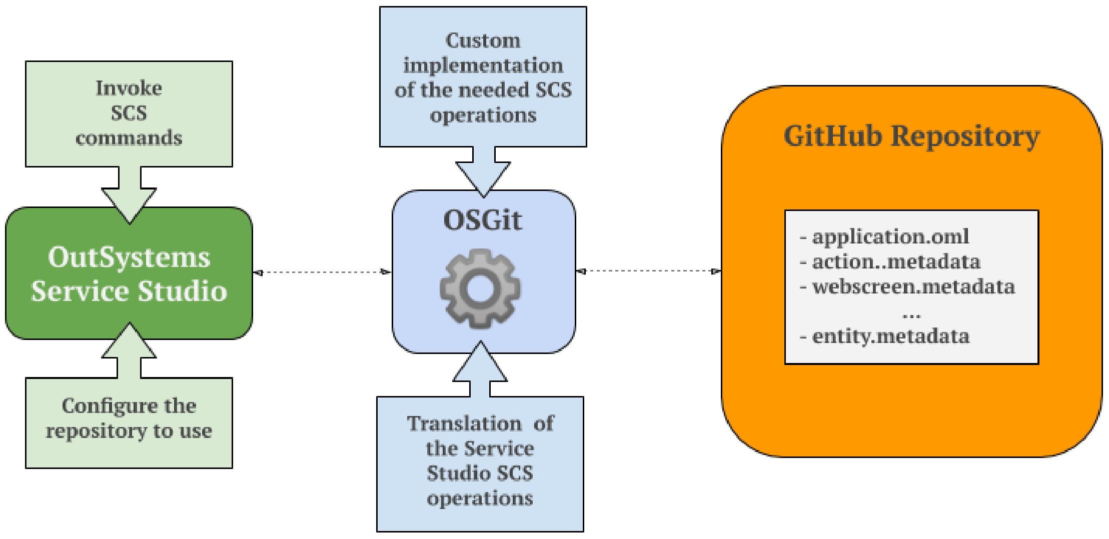

Invoke SCS commands
OutSystems Service Studio
Configure the repository to use
Custom implementation of the needed SCS operations
OSGit
Translation of the Service Studio SCS operations
GitHub Repository
- application.oml
- action..metadata
- webscreen.metadata
...
- entity.metadata

Figure 4.1: Detailed schema of the proposed solution that establishes the communication between the Service Studio and a GitHub repository.

OSGit, as figure 4.1 shows, establishes and coordinates the communications between the OutSystems Service Studio and an external GitHub repository. The SCS operations will be invoked from the Service Studio along-side any necessary configurations. OSGit will then interpret and translate the SCS operations invoked from the Service Studio to native Git operations, which are going to communicate with the external GitHub repository as needed. As previously described, some operations that modify data or require visual constructs to be displayed, need to be addressed differently and cannot be directly translated into native Git operations. Thus the way these operations are used need to be different from the direct usage of the native Git operations mentioned before. This is where OSGit also innovates and is in charge of providing a way to customize the native Git operations calls over visual artifacts that otherwise would not be usable. The external repository, as mentioned, is a GitHub repository created by the user, which will contain the OutSystems application files and the metadata files. OSGit's main purpose is to provide an abstraction layer from what is happening at a lower level, where the Git operation calls and the GitHub communications take place, and any extensions to Git operations are thus made invisible and covered by OSGit.

To enable the abstraction layer, OSGit provides an Application Programming Interface (API), that resides in the core code of the product - in the core of Service Studio - and provides a set of SCS operations, that use Git in its core. These operations provided by our API will not necessarily be a direct match of the chosen Git operations described before. Some operations consist of one or more Git operations. Also, to assist in the implementation of some of the Git operations, the *LibGit2Sharp* [29] library was used. This library provides an API for some of the Git core operations, thus providing a way to avoid using Git operations directly from the command line and parsing their result.

The API, by itself, would not be able to provide any interaction for a user that is using the Service Studio to build his application. So to fill this gap, OSGit also has a simple UI implemented in Service Studio interface, that provides the user with all of the SCS operations available in the API. So to better understand how to incorporate an UI with Service Studio, we produced design mockups of the UI, that shows how the user interacts with the SCS operations, and the type of feedback he/she receives while using them. We also created user stories for every operation available in the UI, that describes the type of user, what they want and why.

Lastly, the technology used to build the back-end section of this integration - the API - was C#, while the front-end part is built with React, TypeScript, and SCSS.

## 4.2 Representing Visual Objects as Text

To build an application with OutSystems, the user must compose and structure a chain of different elements. The application development is done in the Service Studio, and the elements composing the application are displayed on the right-hand side of the screen in the form of a tree, as illustrated in figure 4.2, termed ESpace.

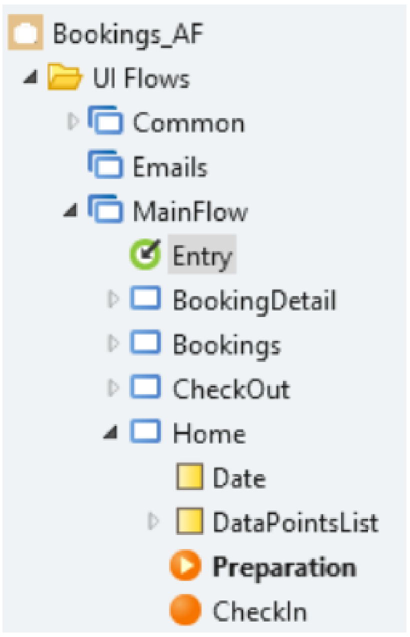


Figure 4.2: The ESpace tree elements of the interface that form the *Bookings* application.

The ESpace tree contains all of the elements that form an application built with OutSystems. For instance, figure 4.2 presents the elements that form the interface of an application. These elements include web screens, like the *Bookings* and *Home* screens, or actions, like the *CheckIn* action that executes a certain logic when the check-in button in the *Home* screen is pressed. These elements are the building blocks of an OutSystems application, and to integrate them with a traditional SCS, we need to represent them in a textural manner.

In order to represent the OutSystems visual elements as text, some attributes of these elements are thus extracted and stored in a file. Figure 4.3 shows an example of the metadata extracted from a web screen and saved in a metadata file.

```
Home|Nodes.WebScreen|TT8tufF_XU6OSMds7eIOgg|efd0ee6e-c8db-410c-84be-a4a954470396
```

Figure 4.3: An example of the metadata information that is stored for each ESpace tree element.

The extracted metadata is composed of four pieces of information that are split by a vertical bar "|". The first piece of information in the metadata example depicted in figure 4.3 is the element's name: *Home*. This name is the name given by the user when the element was created. The next bit of information is the tag name: `Flows.WebScreen`. The tag name indicates the type of each element, for example, in figure 4.2, there are three more web screens: *BookingDetail*, *Bookings*, *CheckOut*. All of these three elements have the same tag names: `Nodes.WebScreen`, because they are all web screens. The tag name will be crucial in grouping the elements of the same type in the metadata files. The third bit of information is very significant. It is the key of the element: `TT8tufF_XU6OSMds7eIOgg`, which will serve the purpose of aiding in the element's searching in the metadata file. The last piece of information is one element property called LastModifiedByCommand: `efd0ee6e-c8db-410c-84be-a4a954470396`. This property indicates if an element was modified, and each time a modification occurs in the element, this property changes its value. Thus, the element's modification will be reflected as a modification in this line.

### 4.2.1 Generating the Metadata Files

The metadata files, contain all the metadata of a given element type, i.e., all of the metadata of the elements that are web screens. For the generation of metadata files, all information referred before - name, tag name, key, and the LastModifiedByCommand property - must be acquired from each element present in the ESpace tree. The ESpace tree is essentially an *n-ary* tree, where the nodes are the elements of the ESpace. Listing 4.1 contains the algorithm used to transverse the tree and to fetch the metadata for each element.

```
GenerateMetadataMap(treeNode, metadataMap) {
    if (treeNode.HasChildren) {
        foreach (child in treeNode.Children) {
            if (child.hasLastModifiedByCommand) {
                tagName = child.TagName;
                List objectsList;
                if (!metadataMap.ContainsKey(tagName)) {
                    objectsList = new List();
                } else {
                    objectsList = new List(metadataMap[tagName]);
                }
                objectsList.Add(
                    child.Name + "|" + child.TagName + "|"
                    + child.Key + "|" + child.LastModifiedByCommand
                );
                metadataMap[child.TagName] = objectsList;
            }
            GenerateObjectsMetadataMap(child, metadataMap);
        }
    }
    return metadataMap;
}
```

Listing 4.1: The algorithm that fetches the metadata information

When traversing the tree, the algorithm visits each node and checks for the existence of the property LastModifiedByCommand. If the node contains the LastModifiedBy-Command property, all of the four pieces of information - name, tag name, key, and the property itself - are saved in a HashMap for further processing. The map's content is already structured very similarly to the structure the individual files have when the metadata information is copied to them. The map's key is an element's tag name, while the value is a list of strings that represent the structured element's metadata, i.e., one line of the file, as seen in figure 4.3. So, for every map entry, there is a list of all metadata elements of the same type, which is how the metadata directory is structured.

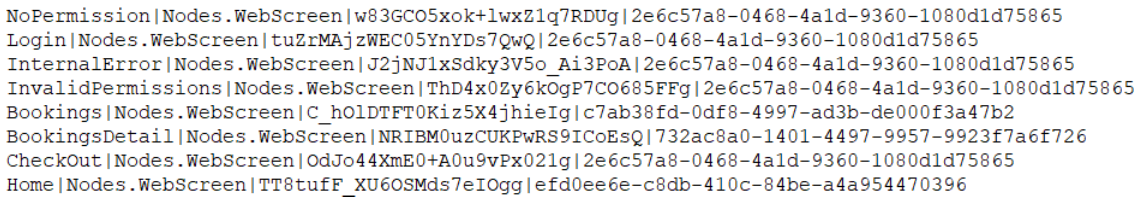
Now that all the metadata information is contained in a structure, we proceed to saving it to the respective files remains to be done. This process has three distinct cases:

1. Nor the file or the element metadata exist;

2. The file exists, but the element metadata has not been added yet;

3. Both the file and the element metadata exist, but the element has been modified.

The first case represents the scenario where a new element is created, and another element of its type does not exist as well. Thus, this is the base case, where both the file and a new file entry containing the element's metadata must be created. The file is created with the same name as the tag name of the element. This helps to maintain an organized file structure, where all of the elements of the same type are in the same file. Notice in figure 4.4, that there are eight metadata entries describing elements of the same type: *Nodes.WebScreen*. All new element's metadata entries are added at the end of the file, so that they are ordered by insertion. This "anchors" each entry to a specific line, which will be useful when using the blame operation, as will be seen later.



```
NoPermission|Nodes.WebScreen|w83GCO5xok+lwxZ1q7RDUg|2e6c57a8-0468-4a1d-9360-1080d1d75865
Login|Nodes.WebScreen|tuZrMAjzWEC05YnYDs7QwQ|2e6c57a8-0468-4a1d-9360-1080d1d75865
InternalError|Nodes.WebScreen|J2jNJ1xSdky3V5o_Ai3PoA|2e6c57a8-0468-4a1d-9360-1080d1d75865
InvalidPermissions|Nodes.WebScreen|ThD4x0Zy6kOgP7CO685FFg|2e6c57a8-0468-4a1d-9360-1080d1d75865
Bookings|Nodes.WebScreen|C_hOlDTFT0Kiz5X4jhieIg|c7ab38fd-0df8-4997-ad3b-de000f3a47b2
BookingsDetail|Nodes.WebScreen|NRIBM0uzCUKPwRS9ICoEsQ|732ac8a0-1401-4497-9957-9923f7a6f726
CheckOut|Nodes.WebScreen|OdJo44XmE0+A0u9vPx021g|2e6c57a8-0468-4a1d-9360-1080d1d75865
Home|Nodes.WebScreen|TT8tufF_XU6OSMds7eIOgg|efd0ee6e-c8db-410c-84be-a4a954470396
```

Figure 4.4: The web screens metadata file, where all application's web screens metadata is stored.

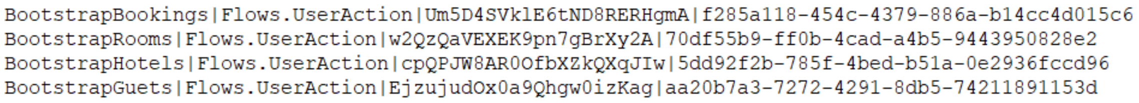
The second case is similar to the first one, but this time there are already elements of the same type in the metadata file, where the new element's metadata is going to be saved. Thus, there is only the need to insert the new element's metadata entry at the end of the file, just like it is done in the first step.

The third and last case happens when both the file and the metadata entry in the file, already exist. This occurs when an existing element is modified, and to be able to point out this modification, the LastModifiedByCommand property changes, and must be reflected in the element's metadata as well. This time the element's metadata is already present in the respective metadata file. This means that only the property LastModifiedByCommand needs to be saved, as the remaining information is the same. Next, the corresponding file needs to be found in the metadata directory. Using the tag name present in the metadata, finding the element inside the file by looking at its key is a direct process. Finally replacing the old LastModifiedByCommand property value for the new one, that indicates that this element was modified, is the last requirement to update this entry in the file.

### 4.2.2 Deleting an Element

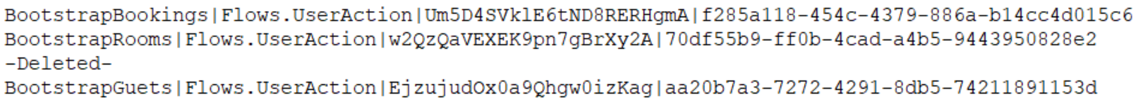
Besides creating and modifying an ESpace tree element, deleting them is also a possibility. So far, to maintain the respective metadata files ordered, element creations pondered on adding its metadata at the end of the file. Moreover, element modifications reflected on changing the LastModifiedByCommand property value in the file which does not move the metadata entry in the file, thus maintaining its original position. However, deleting an element is challenging. When an ESpace tree element is deleted, its metadata entry cannot just be removed from the file, because it disrupts the order of the following metadata entries, as the lines are rearranged in the file. This would produce an erroneous output when using the blame operation on a given element. This incorrect output occurs because the history of the moved element's metadata is lost, as the metadata entry moves to a different line that holds the history of a different element. Moreover, when a metadata entry is created, it cannot be moved away from its first line in the file, so that every element's modification history is safely saved throughout the continuous modifications that may occur on that metadata entry.

The solution to tackle the delete issue is to mark the deleted metadata entry with the text: "-Deleted-". By marking the deleted lines with a different text structure, the order in the file is kept, and no history is mixed or lost between element's, as their metadata entries stay in the same place.



```
BootstrapBookings|Flows.UserAction|Um5D4SVklE6tND8RERHgmA|f285a118-454c-4379-886a-b14cc4d015c6
BootstrapRooms|Flows.UserAction|w2QzQaVEXEK9pn7gBrXy2A|70df55b9-ff0b-4cad-a4b5-9443950828e2
BootstrapHotels|Flows.UserAction|cpQPJW8AR0OfbXZkQXqJIw|5dd92f2b-785f-4bed-b51a-0e2936fccd96
BootstrapGuets|Flows.UserAction|EjzujudOx0a9Qhgw0izKag|aa20b7a3-7272-4291-8db5-74211891153d
```

Figure 4.5: The user actions metadata file content.



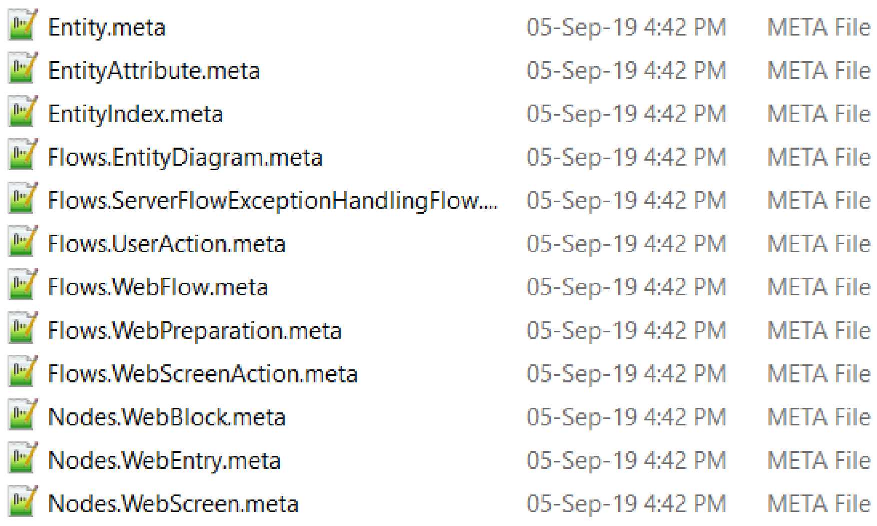
```
BootstrapBookings|Flows.UserAction|Um5D4SVklE6tND8RERHgmA|f285a118-454c-4379-886a-b14cc4d015c6
BootstrapRooms|Flows.UserAction|w2QzQaVEXEK9pn7gBrXy2A|70df55b9-ff0b-4cad-a4b5-9443950828e2
-Deleted-
BootstrapGuets|Flows.UserAction|EjzujudOx0a9Qhgw0izKag|aa20b7a3-7272-4291-8db5-74211891153d
```

Figure 4.6: The user actions metadata file content, with a deleted metadata entry.

In figure 4.5 notice how the third element is deleted during the development of an application, whereas after this process, in figure 4.6 the line of the deleted element is rewritten to indicate that this element ceased to exist, without affecting the global order of the file.

### 4.2.3 The Metadata Directory

The metadata directory is a folder containing all of the metadata files that are associated with an OML. Figure 4.7 shows an example of how the metadata directory looks like after the generation of the metadata files is completed. The metadata files are named after the tag names of the elements of the tree. This way, all elements of the same type are well organized within their file, thus facilitating future searches for a specific element.



| | | |
|---|---|---|
| Entity.meta | 05-Sep-19 4:42 PM | META File |
| EntityAttribute.meta | 05-Sep-19 4:42 PM | META File |
| EntityIndex.meta | 05-Sep-19 4:42 PM | META File |
| Flows.EntityDiagram.meta | 05-Sep-19 4:42 PM | META File |
| Flows.ServerFlowExceptionHandlingFlow.... | 05-Sep-19 4:42 PM | META File |
| Flows.UserAction.meta | 05-Sep-19 4:42 PM | META File |
| Flows.WebFlow.meta | 05-Sep-19 4:42 PM | META File |
| Flows.WebPreparation.meta | 05-Sep-19 4:42 PM | META File |
| Flows.WebScreenAction.meta | 05-Sep-19 4:42 PM | META File |
| Nodes.WebBlock.meta | 05-Sep-19 4:42 PM | META File |
| Nodes.WebEntry.meta | 05-Sep-19 4:42 PM | META File |
| Nodes.WebScreen.meta | 05-Sep-19 4:42 PM | META File |

Figure 4.7: An example of the generated metadata files directory.

## 4.3 The Settings Screen

The Settings screen was the first developed functionality because before being able to start using any Git operations, information like repository paths or user authentications are required. To better understand which path the UI design should follow, the user stories are described in listing 4.2. Also, the design mockup that resulted in the interpretation of the user stories produced, displayed in figure 4.8, will be analyzed further.

```
As an OutSystems developer
I want to authenticate my GitHub account
So that I am able to perform git-based operations on my repository

As an OutSystems developer
I want to specify the URL and the path for my repository
So that I can start performing git-based operations on my repository

As an OutSystems developer
I want to specify my name and email
So that my commits are performed with my information
```

```
As an OutSystems developer
I want to save all the given information
So that I do not need to specify it again
```

Listing 4.2: Settings Screen User Stories.

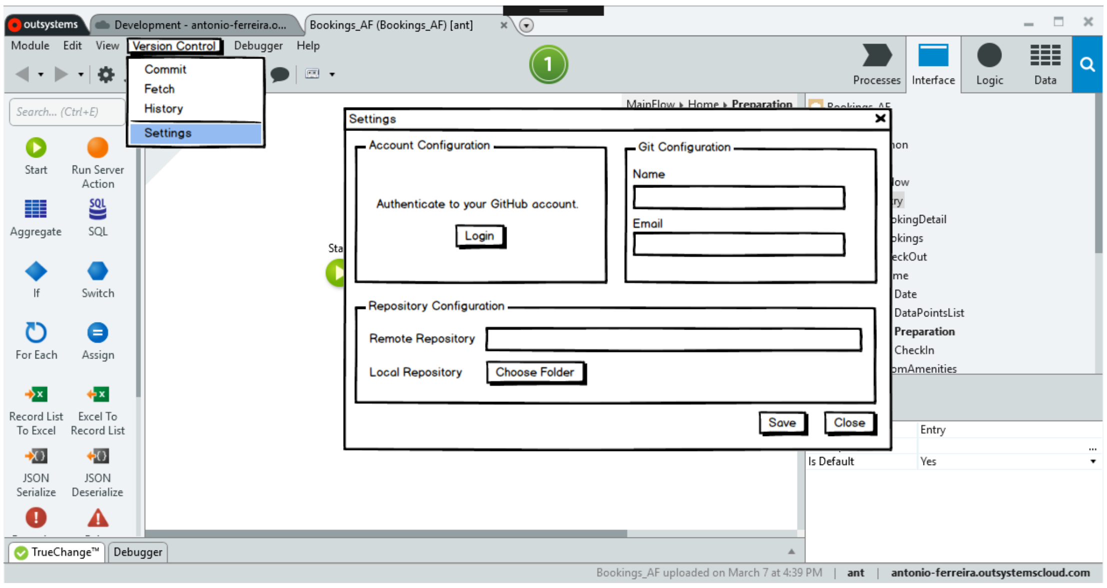


Figure 4.8: The mockup of the Settings menu, from where the users can configure the Git repository and authenticate to GitHub.

Firstly, the user must authenticate with a remote repository host, which in this case is GitHub, to be able to interact with a remote repository. Secondly, the GitHub repository link must be provided to perform certain Git operations, namely the clone operation. Also, a local folder path must be provided, so that a local version of the remote repository can be stored. Lastly, the user must provide its name and email, because this information is required in some Git operations, such as commits and merges, as a way to identify the person who performed them. Taking this into consideration, the last thing that is left to do is to save the required information for the remaining operations of our system to be able to use and give the user an environment where he/she will be ready to work. It also means that a copy of the remote repository must be brought to a local folder.

### 4.3.1 The Account Configuration Box

In this section, as shown on the left-hand side of the "Settings" screen in figure 4.8, the user can authenticate to GitHub with his account. Recurring to the tool *Git Credential Manager for Windows* [18], helped keeping the communications with GitHub safe, as the authentication process involves password exchanges. This tool is a secure Git credential storage, which handles all of the network communications with GitHub. It also provides a local secure storage for Git passwords so that the user does not need to log in every time he/she intends to use our system.

When pressed, the *Login* button opens the *Git Credential Manager* authentication window, as seen in figure 4.9. This is done via a simple command:

```
git credential-manager fill
```

This command waits for an input, that is the repository host to authenticate with, which is GitHub. Therefore, the provided input is as follows: "`host=https://github.com`".

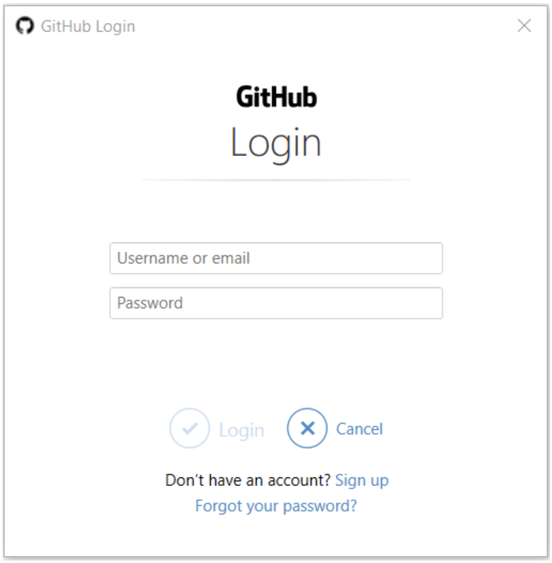


Figure 4.9: The *Git Credential Manager* GitHub authentication window.

Now, the user can proceed with the authentication process by filling the fields in figure 4.9, thus authenticating himself to GitHub.

### 4.3.2 The Git Configuration Box

The Git Configuration Box, as seen in the right-hand side of the "Settings" screen in figure 4.8, allows the user to give information about himself, namely his name and email, so that Git operations like merges and commits, can identify the person that performs them. These two fields - name and email - are simple text inputs, with the possibility to be filled automatically. The automation process occurs after the user successfully authenticates with GitHub. After the authentication process, a call to the GitHub API is made, requesting information with the following command:

```
curl -u username:token https://api.github.com/user
```

The *token* stands for the secured token secret that was generated and stored in the *Git Credential Manager* for the current user. This information may or may not be already present in the user's GitHub account. In the cases that it is not, the respective field stays blank for the user to fill.

### 4.3.3 The Repository Configuration Box

In this segment of the screen, as seen in the bottom of figure 4.8, all the repository related configurations take place. The first field, as the name suggests, receives the link to the Git repository where the user will be working on. The second field, represented by the button *Choose Folder*, is intended for allowing the user to browse the file system, and choose the desired folder where the local version of his repository will be held.

### 4.3.4 Saving settings

The *Save* button, on the bottom right-hand side of figure 4.8, saves all the information specified in the boxes described above - username, name, email, local and remote repository paths - in the persistent settings provided by the Service Studio, so that whenever the user closes or opens the Service Studio, all of this information persists, and is not required to be inserted again. Also, this button only becomes available if all of the fields are correctly filled.

The most important thing the *Save* button does is cloning the remote repository to the local folder specified by the user in the Git Configuration Box. This step recurs to the git clone operation with the provided remote repository path as input. The saving step also has the particularity to verify if an OML is already present in the repository. If this is true, then the OML is automatically opened in the Service Studio, ready for the user to work. After the saving is complete, the user is left in a safe state, where all the information that he/she provided is saved, and the repository is ready to be used.

Besides showing how the implemented settings screen UI is displayed, figure 4.10 also shows how the settings screen looks like when all of its fields are filled. Notice that after a successful authentication, the Account Configuration Box changes, so that it displays a new button where the user can perform the logout operation. This operation deletes the credentials stored in the *Git Credential Manager* that were previously created when the user authenticated with GitHub. The credentials deletion operation is done via the following command:

```
git credential-manager reject
```

As previously described, this command also expects the same input indicating which repository host we are referring to.

Lastly, the field *Local Repository*, after successfully choosing a path, this path is now shown before the *Choose Folder* button, which now changed to the *Change* button that allows to change the previously selected path.

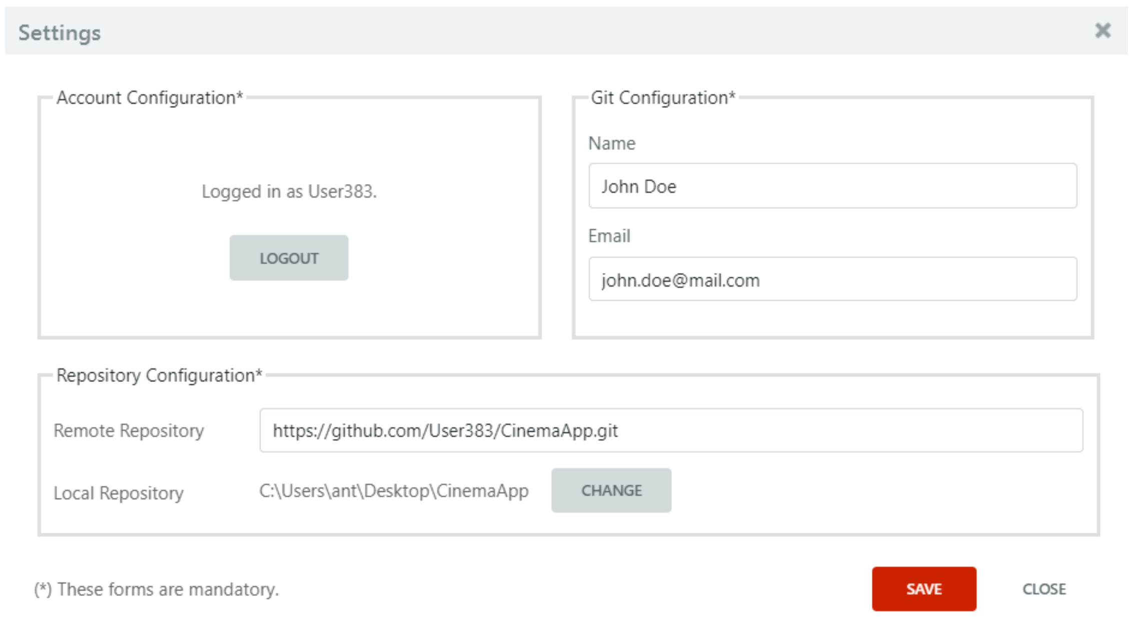


Figure 4.10: The implemented version of the Settings Screen UI.

## 4.4 The Commit Operation

The commit operation was the following developed functionality. For the commit operation, user stories were also created and are displayed in listing 4.3, where they provide the global vision over the functionalities of this operation. For this case, the only user interaction is when the user must write a commit message, receive feedback while the commit is in progress and when the commit completes. Thus, one new design mockup arises: the commit message screen, shown in figure 4.11. In this message screen, the user is going to input his commit message that reflects the changes he/she performed with his commit.

```
As an OutSystems developer
I want to perform a commit
So that I can save my current work on my repository repository

As an OutSystems developer
I want to write my commit message
So that others can understand what has changed with my commit

As an OutSystems developer
I want to merge the changes present in the remote repository
So that my local repository is up-to-date and my commit concluded

As an OutSystems developer
```

```
I want to solve the conflicts that appeared with the merge operation
So that I can successfully  merge the remote and local revisions of
my project, thus completing my commit
```

Listing 4.3: Commit Operation User Stories.

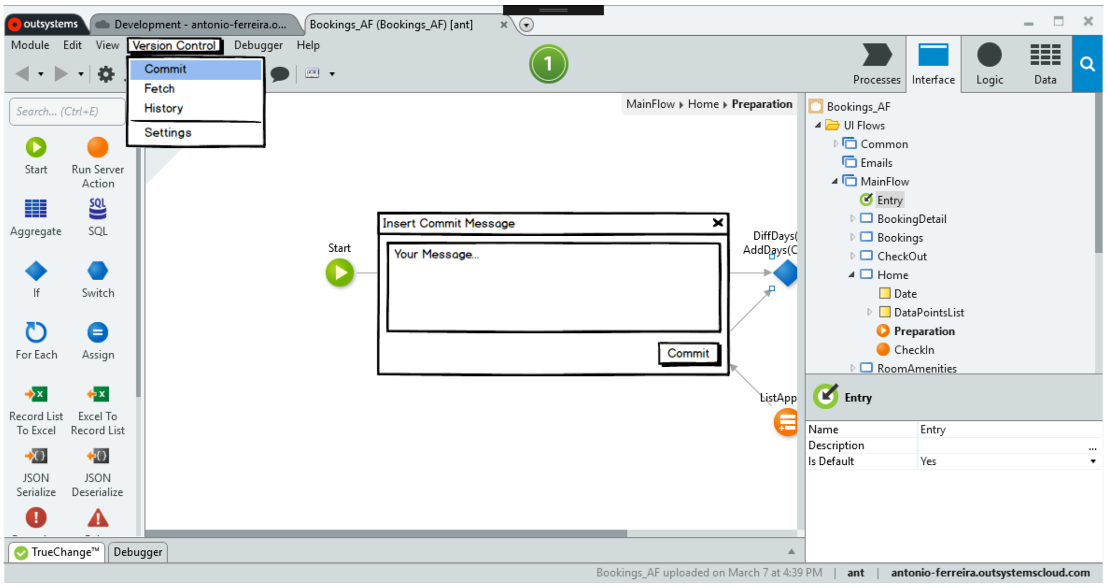


Figure 4.11: The commit message input, for the user to introduce his commit message.

The commit operation provided in our solution is different from the commit operation that is present in Git. The commit operation available in Git only saves the current modifications to the local repository while creating a new version. In our solution, the commit operation also saves the modifications in the remote repository, i.e., the commit operation in our solution uses the Git commit operation to save the modifications locally, and then utilizes the Git push operation to send and save them in the remote repository as well. This approach is preferred because it tends to minimize potential conflicts in the code that is made by inexperienced users. In this situation where the user is not used to a SCS workflow, in the scenario where the commit operation is the same as Git's, the user may tend to spend longer periods without saving his modifications to the remote repository. This introduces a high-cost conflict resolution effort when he/she tries to save all of the local modifications in the remote repository, which may already have advanced work performed by other team members.

The commit and push workflow, however, raises a new problem. When the Git push operation takes place, the remote repository might have modifications that the user does not have locally. This means that both the remote and local versions have to be merged and conflicts may arise. Thus, two Git operations must integrate the solution's commit operation: Git pull and merge, the first is to bring the remote version to the local repository and the latter needed to merge the local and remote versions.

### 4.4.1 The Operation Implementation

Regardless of the SCS, performing a commit, only makes sense if any pending modifications need to be saved. Thus, checking for pending modifications is required before allowing the user to try to commit his work. Figure 4.12 shows the commit operation flow.

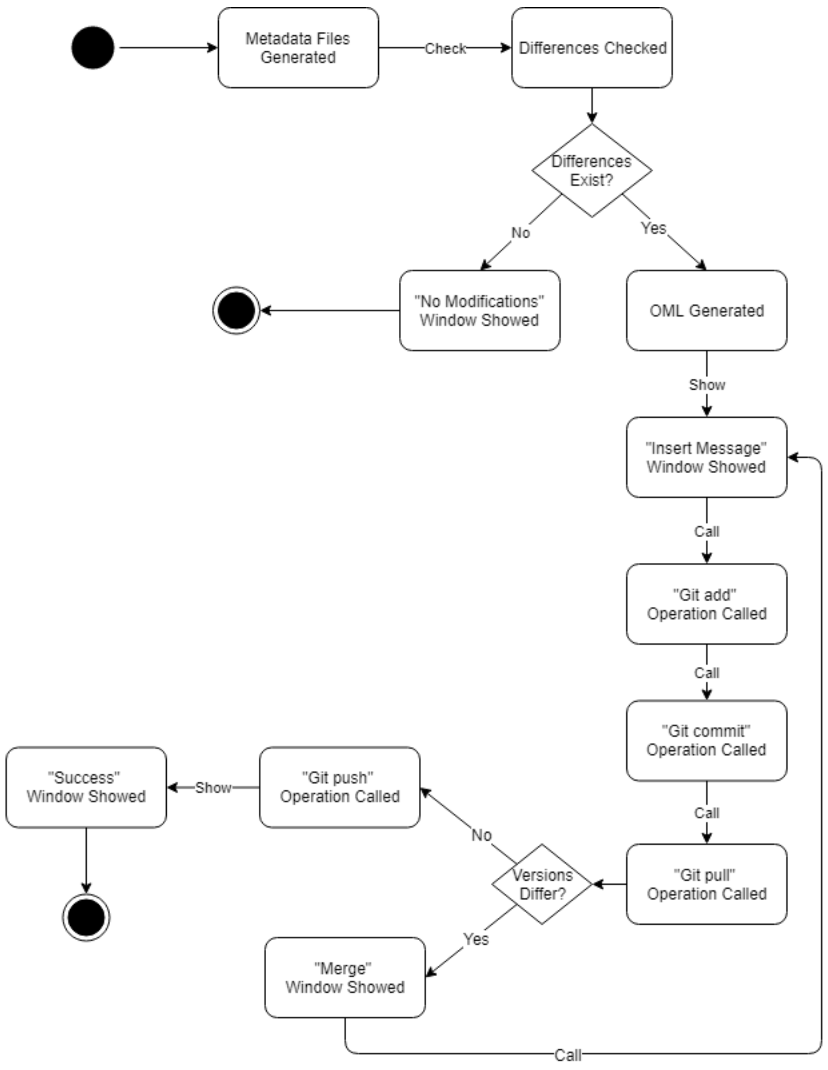


Figure 4.12: The state diagram of the commit operation.

The first step is to generate the metadata files. It is by checking for modifications - with the Git status operation - in these files, that states whether or not the commit should proceed. If there are not any modifications detected, a simple dialog informing the user that there are no pending modifications to be saved is shown. On the other hand, if modifications are detected, the next step before committing is to generate the application file, the OML, so that it can be saved in the remote repository. This step involves saving the current OML opened in the Service Studio to the local repository folder, specified in the settings screen. The following step is to perform the actual commit. This procedure, as seen in figure 4.12 requires three steps: displaying the commit message input, followed by calling the Git add operation and lastly performing the Git commit operation. Firstly a small input box, as shown in figure 4.13 is displayed. After capturing the user commit message, all the necessary information required for the commit operation is gathered. The next step is to call the Git add operation because, before using the Git commit operation, the existing modifications must be staged. Only staged modifications are considered in the commit operation. So, after this step, the Git commit operation can finally be called with the commit message previously inserted by the user.

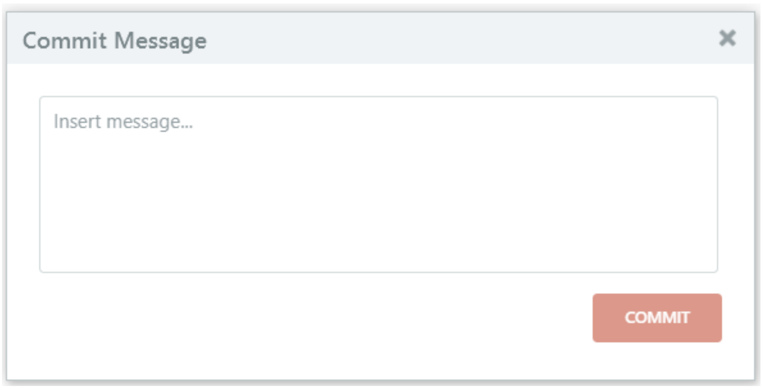
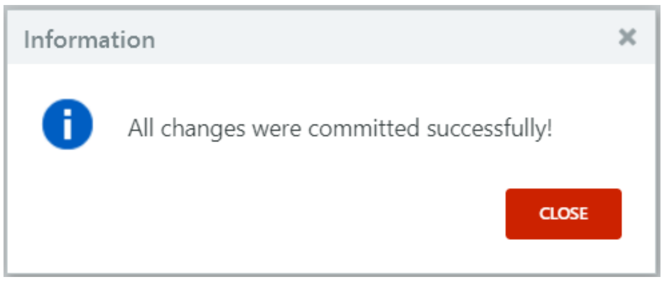


Figure 4.13: The implemented commit message input.

The commit section of the commit and push workflow is now complete. By proceeding to the push part of the workflow, figure 4.12 shows that the next step is to use the Git pull operation. This operation is needed because it checks for modifications in the remote repository and brings them to the local repository if they exist. Also, performing a Git push operation requires that the local repository is updated with the remote one. After the Git pull operation is executed, two paths emerge. In the first path, the local repository is up-to-date with the remote repository, and the flow can proceed to push the local modifications to the remote repository. In the second path, the remote repository contains work that the local repository does not have yet, thus requiring both the remote and local versions to be merged, before pushing the local modifications to the remote repository. In the first path, where the local repository is up-to-date with the remote repository, the flow can proceed to execute the Git push operation to save the local modifications on the remote repository. After this step, a feedback dialog indicating that the local modifications were stored in the remote repository is shown, as seen in figure 4.14.

In the second path, where both the local and remote repositories have different versions, there is the need to merge both versions in order to be able to save the final merged version onto the remote repository. Following figure 4.12, if the versions differ, the flow continues to the display of the merge window. The merge operation used to merge two versions of an OML is not the merge operation offered by Git, because the text-oriented merge algorithms do not work with OMLs. OutSystems had already produced a merge algorithm that works with the OML file architecture, which is the algorithm that is used


Information

All changes were committed successfully!

Figure 4.14: The implemented commit feedback dialog.

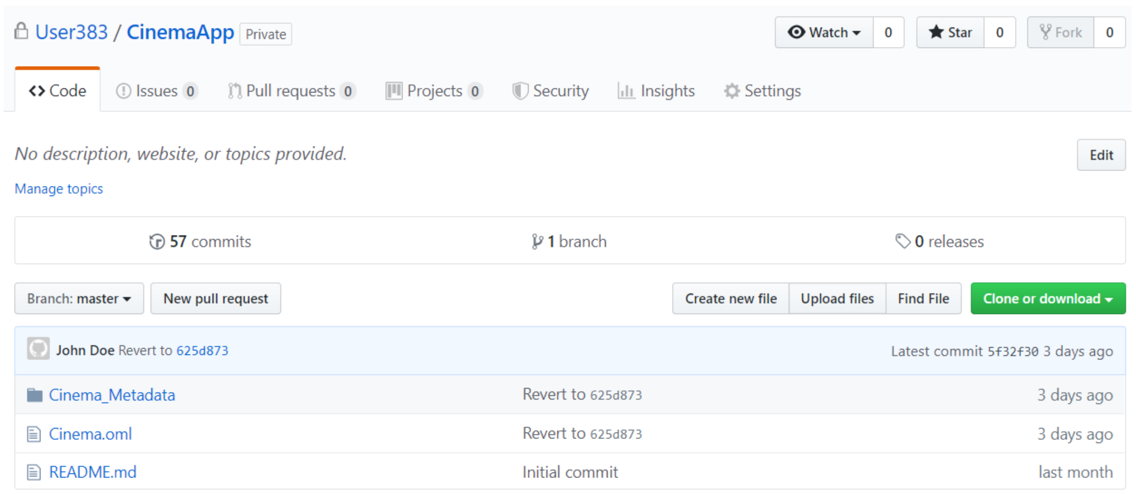
here to replace the Git merge. This algorithm already has an interface from where the user can compare and merge two versions of an OML, as seen in figure 3.1. On the left-hand side of the merge window, there is the user's local version, and on the right-hand side is the version present in the remote repository. Notice that the merge algorithm needs two OMLs to work. One OML is already present in the Service Studio, which is the user's local version. So, to get the OML's version present in the remote repository, the last commit ID is obtained and followed by fetching the OML version for that commit, via the following command:

```
git show lastCommitID:application.oml
```

After the user merges both versions of the OML, the local repository is left with new modifications that resulted from the merging process. But before continuing figure's 4.12 flow, the metadata files, and the OML must be generated again and stored in the local repository. After the latter has been concluded, the new modifications are ready to be committed again. Thus the figure's 4.12 flow, returns to the previous step of introducing a new commit message for the newly merged modifications. Although this time, the commit message - "`Merge branch "master" of https://github.com/user/repository.git`" - is automatically filled for the user, as the previous modifications resulted from merging two versions of the same application. From now on, all of the previous steps, involving staging the modifications, committing them locally and checking for updated remote versions, are repeated in a loop until the remote repository versions match the local version, and the local modifications are pushed to the remote repository, as seen in figure 4.12.

### 4.4.2 The Repository State

After the user commits a new version of his work to the remote repository, the application's OML and the generated metadata files are the only contents that are sent to the repository, as we can see in figure 4.15. The metadata files are grouped in a folder named after the application's OML.


User383 / CinemaApp Private

No description, website, or topics provided.

Manage topics

57 commits · 1 branch · 0 releases

John Doe Revert to 625d873 — Latest commit 5f32f30 3 days ago

| | | |
|---|---|---|
| Cinema_Metadata | Revert to 625d873 | 3 days ago |
| Cinema.oml | Revert to 625d873 | 3 days ago |
| README.md | Initial commit | last month |

Figure 4.15: The GitHub repository content.

## 4.5 The Blame Operation

Following the footsteps of the previous functionalities, user stories were created for this operation, which are presented in listing 4.4. Like the previous functionality, the user stories helped to define the UI and understand the user interactions with this operation. The blame operation UI design mock is shown in figure 4.16.

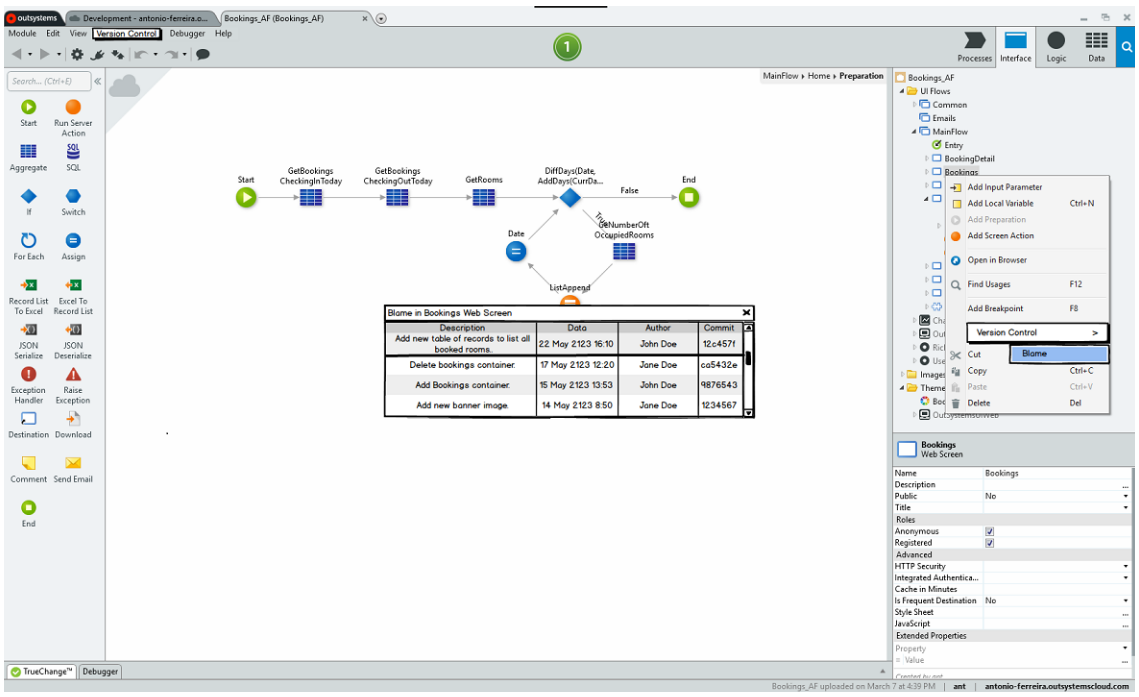
```
As an OutSystems developer
I want to see the last person that modified the Home web screen
So that I can warn him of an important feature that is missing
in the screen

As an OutSystems developer
I want to see the last person that modified the Settings web screen
So that I can see the commit message and perform an appropriate code
review of the changes made to the screen

As an OutSystems developer
I want to see the last person that modified the FetchUsers action
So that I can see the differences between his version and the version
I committed yesterday
```

Listing 4.4: Blame Operation User Stories.

The blame operation that our solution provides is mostly implementing the same idea as the blame operation offered by Git. The Git blame operation annotates each line of code, which are lines of text, with information about the last person who modified it. Extracting the blame idea to the paradigm of the visual language can be challenging. The problem is that there are no lines of text when programming with OutSystems that can be interpreted by humans, that would otherwise be interpreted with text-based programming languages.



Blame in Bookings Web Screen

| Description | Data | Author | Commit |
|---|---|---|---|
| Add new table of records to list all booked rooms.. | 22 May 2123 16:10 | John Doe | 12c457f |
| Delete bookings container. | 17 May 2123 12:20 | Jane Doe | ca5432e |
| Add Bookings container. | 15 May 2123 13:53 | John Doe | 9876543 |
| Add new banner image. | 14 May 2123 8:50 | Jane Doe | 1234567 |

Figure 4.16: The Blame result for the "Bookings" web screen.

The OutSystems files, the OMLs, are binary files, and directly applying a blame over these files in a human-readable manner cannot be done correctly. Moreover, building applications with OutSystems is about composing and structuring elements - like actions or web screens - that correlate with each other to form an application. It is very different from writing a line of code that executes a particular action. So, defining what applying a blame means in the OutSystems language is critical, as it cannot be applied directly to OMLs. Furthermore, our vision for the blame operation goes further than the blame operation offered by Git. The Git blame operation shows the last modification on a given line of text, and our intent is to show the full modification history of a given element. Despite being possible to use Git blame to see any prior version of a line of text, we have to specify the commit ID of the revision we want to blame. This would not be optimal, because if we want to see the full modification history, we have to get every commit ID and re-execute the Git blame operation for each one of those commit ID's. To this extent, the best operation to use is not the Git blame operation, but instead an operation that simulates the Git blame behaviour and that fetches the full modification history of an element. This operation is a parameterized version of the Git log operation.

As the "lines of code" are the building blocks of an application build with a text-based programming language, the ESpace tree elements are the building blocks of an OutSystems application as depicted in figure 4.2. These elements are thus the targets to apply the blame operation. Hence the OSGit's blame operation main idea is to blame an ESpace tree element: for instance figure's 4.2 *Bookings* screen, and retain a history of the last modifications made to that screen, as represented by the mock in figure 4.16.

Lastly, to make use of Git log operation (the detailed version of Git blame), as we are before a text-oriented SCS, some form of text representation is required. This is where the metadata files are used. As the building blocks of an OutSystems application are the elements of the ESpace tree, by using the metadata to represent them in a text manner, the blame operation over each element can work well.

### 4.5.1 The Operation Implementation

The blame operation as described before, is applied to the metadata files, which have the textual representation of the elements in the ESpace tree - figure 4.2. So, the Git log operation is applied to one line of one metadata file, which has the metadata representation of a given ESpace tree element, also containing its full modification history. This works because this metadata line changed as the respective element was being modified, and thus the element's history is resides in the modification history of its metadata line.

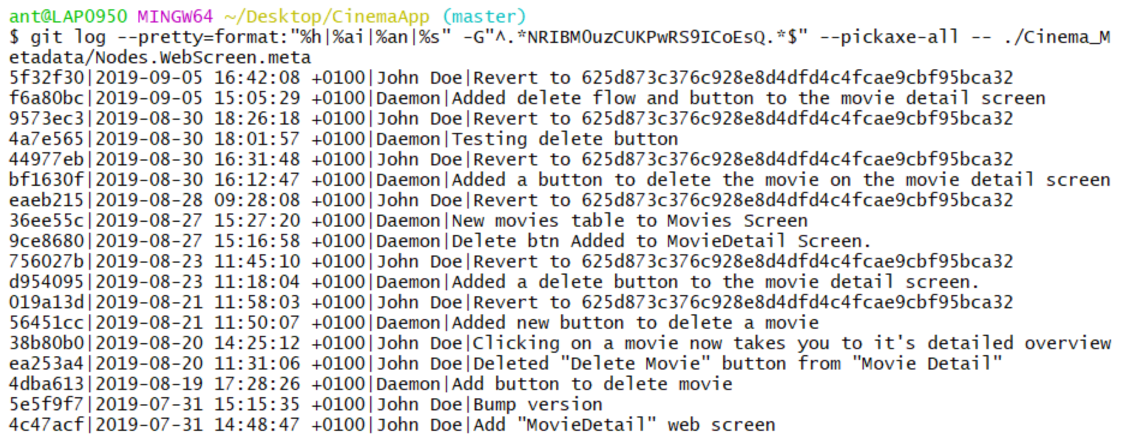
Similarly to the Git blame operation, which extracts information that includes the commit ID, commit date, and the name of the person who did it, the parameterized version of the Git log operation, will also extract the same information. Furthermore, the Git log cannot be applied to the whole file, as there is only the need to retrieve information about one line of the file that corresponds to one ESpace tree element. Thus the parameterized Git log operation is as follows [17]:

```
git log --pretty=format:"%h|%ai|%an|%s" -G"^.*key.*$" --pickaxe-all --filePath
```

The first parameter of the command, `--pretty=format:"%h|%ai|%an|%s"` is the output formating options, which indicates that each line of the output presents the commit ID, commit date, the commit's author name, and the commit message, separated by a vertical bar "`|`". The following two parameters dictate the search method, i.e., to find all modifications that contain the regular expression specified after the "`-G`" option. This regular expression contains the keyword *key*, which is the key of the element's metadata. Lastly, the "`--filePath`" argument, specifies in which file, the regular expression lookup takes place.

To know which values to assign to the *key* and the *filePath* parameters, we need to find the corresponding metadata file and element's key, respectively. Taking figure 4.16 as an example, let us suppose the user utilizes the blame operation on the *Bookings* web screen as shown. The first thing to do is to retrieve the tag name of the element the user clicked, i.e., the tag name of the *Bookings* web screen. With the tag name, finding the metadata file where the *Bookings* web screen metadata is stored is a direct process, as the file has the same name as the tag name. The following step is to obtain the element's key. Also, the key is already known as it is associated with the *Bookings* web screen that the user clicked to perform the blame operation.

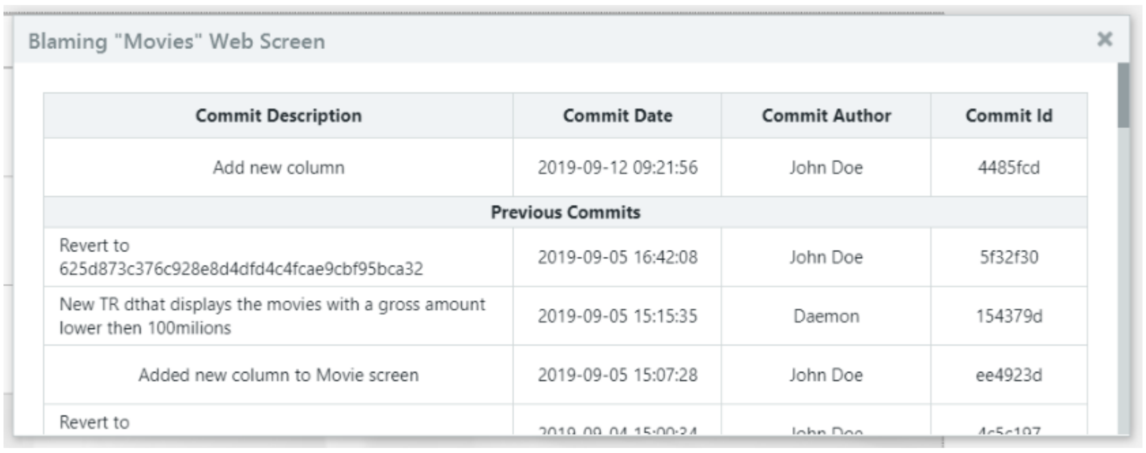
The element's key and the file name are then used in the parameters of the command described above. Its output will be multiple strings, with each string containing the four bits of information request in the parameter `--pretty=format:"%h|%ai|%an|%s"`: commit ID, commit date, the commit's author name, and the commit message, as shown in figure 4.17. Each string essentially contains the said information pieces about a commit that modified the *Bookings* web screen in a given point in time. Lastly, these strings will be stored in an array that will be used in the next step of the blame operation.



```
ant@LAP0950 MINGW64 ~/Desktop/CinemaApp (master)
$ git log --pretty=format:"%h|%ai|%an|%s" -G"^.*NRIBM0uzCUKPwRS9ICoEsQ.*$" --pickaxe-all -- ./Cinema_M
etadata/Nodes.WebScreen.meta
5f32f30|2019-09-05 16:42:08 +0100|John Doe|Revert to 625d873c376c928e8d4dfd4c4fcae9cbf95bca32
f6a80bc|2019-09-05 15:05:29 +0100|Daemon|Added delete flow and button to the movie detail screen
9573ec3|2019-08-30 18:26:18 +0100|John Doe|Revert to 625d873c376c928e8d4dfd4c4fcae9cbf95bca32
4a7e565|2019-08-30 18:01:57 +0100|Daemon|Testing delete button
44977eb|2019-08-30 16:31:48 +0100|John Doe|Revert to 625d873c376c928e8d4dfd4c4fcae9cbf95bca32
bf1630f|2019-08-30 16:12:47 +0100|Daemon|Added a button to delete the movie on the movie detail screen
eaeb215|2019-08-28 09:28:08 +0100|John Doe|Revert to 625d873c376c928e8d4dfd4c4fcae9cbf95bca32
36ee55c|2019-08-27 15:27:20 +0100|Daemon|New movies table to Movies Screen
9ce8680|2019-08-27 15:16:58 +0100|Daemon|Delete btn Added to MovieDetail Screen.
756027b|2019-08-23 11:45:10 +0100|John Doe|Revert to 625d873c376c928e8d4dfd4c4fcae9cbf95bca32
d954095|2019-08-23 11:18:04 +0100|Daemon|Added a delete button to the movie detail screen.
019a13d|2019-08-21 11:58:03 +0100|John Doe|Revert to 625d873c376c928e8d4dfd4c4fcae9cbf95bca32
56451cc|2019-08-21 11:50:07 +0100|Daemon|Added new button to delete a movie
38b80b0|2019-08-20 14:25:12 +0100|John Doe|Clicking on a movie now takes you to it's detailed overview
ea253a4|2019-08-20 11:31:06 +0100|John Doe|Deleted "Delete Movie" button from "Movie Detail"
4dba613|2019-08-19 17:28:26 +0100|Daemon|Add button to delete movie
5e5f9f7|2019-07-31 15:15:35 +0100|John Doe|Bump version
4c47acf|2019-07-31 14:48:47 +0100|John Doe|Add "MovieDetail" web screen
```

Figure 4.17: The output of the used blame command.

The following step of the blame operation is to display the fetched blame history - the array of strings - to the user, as shown in the mock in figure 4.16. Figure 4.18 shows the implemented blame UI. As it can be observed, there is a table-like display of the information fetched by the command described earlier. Each line of the table maps to each string of the provided array that contains the blame information. Notice that the first row always shows the most recent modification made to the respective ESpace element. Furthermore, all of the remaining rows are ordered by commit date, i.e., by modification date.



Blaming "Movies" Web Screen

| Commit Description | Commit Date | Commit Author | Commit Id |
|---|---|---|---|
| Add new column | 2019-09-12 09:21:56 | John Doe | 4485fcd |
| **Previous Commits** | | | |
| Revert to 625d873c376c928e8d4dfd4c4fcae9cbf95bca32 | 2019-09-05 16:42:08 | John Doe | 5f32f30 |
| New TR dthat displays the movies with a gross amount lower then 100milions | 2019-09-05 15:15:35 | Daemon | 154379d |
| Added new column to Movie screen | 2019-09-05 15:07:28 | John Doe | ee4923d |
| Revert to | | | |

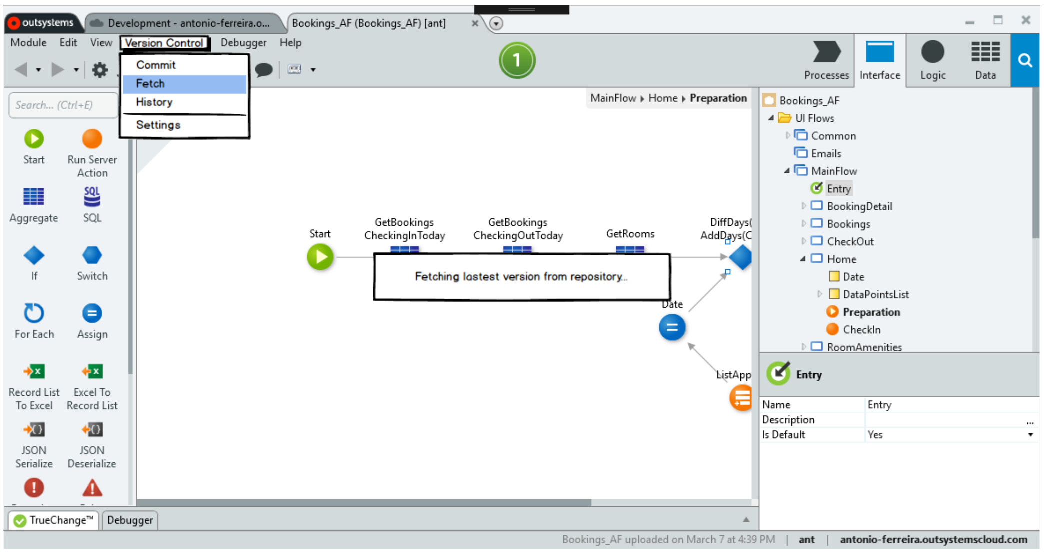
Figure 4.18: The implemented blame screen.

## 4.6 Additional Operations

Two of the proposed operations in section 3.7.2: the fetch and history operations, were not implemented in the final proof of concept due to time constraints. However, some work for these two operations was developed and is presented next.

### 4.6.1 The Fetch Operation

The fetch operation is required so that users are able to bring the changes stored in the remote repository to the local repository. This operation is useful if the user wants to check if there is advanced work in the remote repository that he/she does not have in his local repository. As described throughout chapter 4, the initial steps before implementing the featured operations in the proof of concept, were to write user stories to understand the usability of the operation in hand and to produce design mocks of the featured operation screen. As presented in listing 4.5, the produced user stories for the fetch operation are shown. In figure 4.19, a simple feedback message that must be shown to the user while executing the fetch operation, is displayed. The next steps would be to implement this feature, taking into consideration the work that has already been done.



Fetching lastest version from repository...

Entry

| | |
|---|---|
| Name | Entry |
| Description | |
| Is Default | Yes |

Bookings_AF uploaded on March 7 at 4:39 PM | ant | antonio-ferreira.outsystemscloud.com

Figure 4.19: The Fetch operation feedback.

```
As an OutSystems developer
I want to obtain the changes in the remote repository
So that I have an updated local repository

As an OutSystems developer
I want to resolve any merge conflicts while fetching the remote work
So that I can complete the fetching process
```

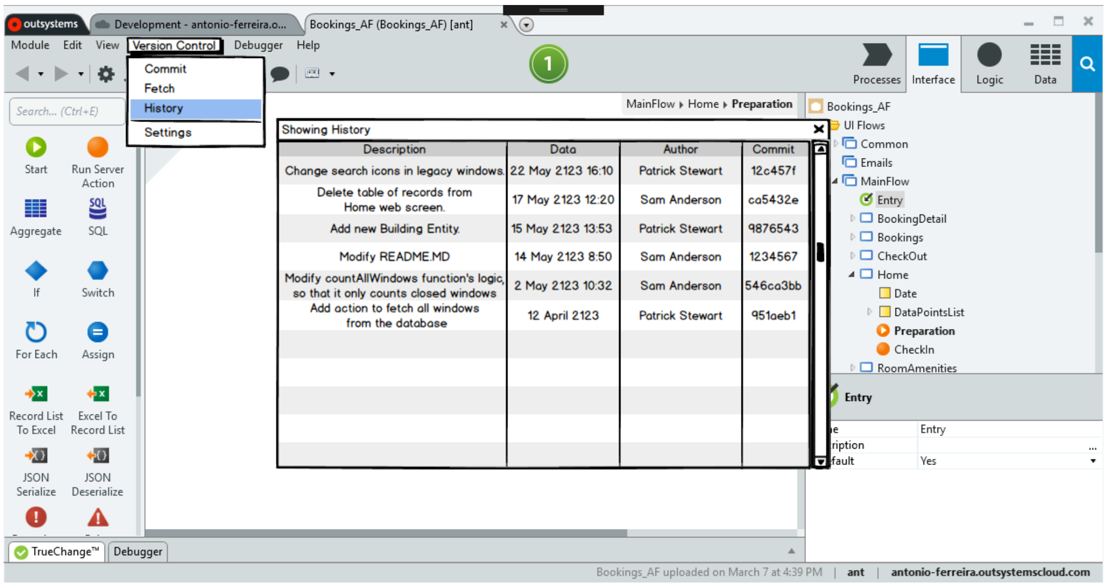
Listing 4.5: Fetch Operation User Stories.

It must be considered that possible conflicts might occur when the user fetches the changes in the remote repository. So, the merge operation must be included in the fetch workflow if such conflicts are detected. Lastly, this operation would be easy to implement, because a part of it is already implemented in the fetching process of the commit operation of OSGit.

### 4.6.2 The History Operation

The history operation intends to display the full repository history to the user. Information such as commit messages, commit ID's, commit author's names and commit dates, should at least be included in the history provided.

This operation follows the same path as the fetch operation described above. The user stories and design mocks for this operation, were also produced and are displayed in listing 4.6 and figure 4.20 respectively.



Bookings_AF (Bookings_AF) [ant]
Module Edit View Version Control Debugger Help
Commit
Fetch
History
Settings
1
Processes Interface Logic Data
MainFlow ▸ Home ▸ Preparation
Bookings_AF uploaded on March 7 at 4:39 PM | ant | antonio-ferreira.outsystemscloud.com

Showing History

| Description | Data | Author | Commit |
|---|---|---|---|
| Change search icons in legacy windows. | 22 May 2123 16:10 | Patrick Stewart | 12c457f |
| Delete table of records from Home web screen. | 17 May 2123 12:20 | Sam Anderson | ca5432e |
| Add new Building Entity. | 15 May 2123 13:53 | Patrick Stewart | 9876543 |
| Modify README.MD | 14 May 2123 8:50 | Sam Anderson | 1234567 |
| Modify countAllWindows function's logic, so that it only counts closed windows | 2 May 2123 10:32 | Sam Anderson | 546ca3bb |
| Add action to fetch all windows from the database | 12 April 2123 | Patrick Stewart | 951aeb1 |

Figure 4.20: The History result, which shows the full commit history of the repository.

```
As an OutSystems developer
I want to observe all the commits in the repository
So that I can see who performed them and why

As an OutSystems developer
I want to observe the commit done by Paul last Monday
So that I can see the differences between his version and my
current version
```

Listing 4.6: History Operation User Stories.

As shown in figure 4.20, it is intended to shown each commit information in a table-like display, where each line of the table could be clicked, and take the user to the compare and merge screen, figure 3.1, so that the user could compare or revert any changes at any point in history. However, this operation is not essential, because GitHub already features a commit history view.

Chapter 5

# Evaluation

This chapter introduces and discusses the evaluation step. We performed usability tests in order to validate our solution, and which execution and feedback will be approached next.

## 5.1 Experiment Design

Usability tests measure user performance in a given piece of software. This interaction is measured by covering user productivity, goals accomplished, and satisfaction, which are the key aspects that are going to evaluate our solution [13].

There are several ways we could conduct this study. One would be by evaluating the participants' performance in a version of the Service Studio with OSGit - scenario A - and in another version without OSGit - scenario B. The participants would then perform a series of tasks over scenario A and repeat them for scenario B as well. At the end, we would compare the usability of both scenarios. In a second approach to measure usability, the participants' performance is only evaluated in one version of the Service Studio with OSGit, where they will also perform a defined series of tasks in this scenario. We chose the second approach because the first one is more complex and would take more time to complete, which is an issue due to the industrial environment we are conducting our study, where the number of participants and their available time are rather limited.

The second case consisted of providing a Service Studio version with the OSGit integration. A new build containing the integration was generated and installed in a virtual machine hosted by Amazon Web Services. Using a virtual machine to hold the Service Studio version with OSGit, avoids the installation setup time that the participants would have to spend. It also makes it easier for us to control the environment where the participants are doing the exercise and prepare it accordingly to speed up the experiment duration. The following step consisted of handing over to the participants a set of tasks that they will perform by using OSGit's Service Studio integration. We called it the "SCS User Acceptance Exercise" and it is available in appendix B. All the tasks that the participants are asked to perform in this exercise sheet consist of using every operation provided in OSGit and their respective workflows. For example, in the commit operation, there are two possible scenarios: the user commits without conflicts, and the user commits after solving the conflicts. These two scenarios were simulated in the experiment, thus the similarity of the first and third points of the "Onto the Exercise" section of appendix B.

At the end of the exercise, the participants were asked to fill the post-test questionnaire displayed in appendix A. This questionnaire features some basic demographic questions, a section of questions regarding the utility of OSGit, also with open-ended questions, and the last section featuring the SUS questionnaire [6].

The experiment was built to have a maximum duration of thirty minutes, given the industrial environment where the study is being conducted.

## 5.2 Participants

The participants to conduct this study need to be people that use OutSystems daily. These people are the ones that are capable of giving the most accurate and reasonable feedback because of their experience working with the Service Studio, thus being able to provide further insights about the usability of our tool. To this extent we asked for volunteers inside OutSystems, i.e., OutSystems employees that work with the platform daily. In total, we could find ten available participants.

| Participant Number | Gender | Age | OutSystems Experience (Years) | Academic Background |
|---|---|---|---|---|
| 1 | Male | 30-40 | 10 | Computer Science |
| 2 | Male | 30-40 | 10 | Computer Science |
| 3 | Male | 30-40 | 3 | Computer Science |
| 4 | Male | 30-40 | 5 | Computer Science |
| 5 | Female | 30-40 | 3 | Environmental Engineering |
| 6 | Male | 20-30 | 3 | Computer Science |
| 7 | Male | 30-40 | 7 | Computer Science |
| 8 | Male | 40-50 | 6 | Computer Science |
| 9 | Female | 20-30 | 2 | Computer Science and Business Management |
| 10 | Female | 20-30 | 3 | Computer Science |

Table 5.1: Demographic information of the ten participants.

As depicted in table 5.1, the participant's pool varied in ages between 20 to 50 years old, with 70% males and 30% females. Also, from a technical point of view, we had people from various teams inside OutSystems, that also differed in the academic background, where two people had bachelors in environmental engineering and business management, and the remaining had a computer science background. Another important aspect was the familiarity with the OutSystems Language, i.e., we had participants ranging from 2 to 10 years of experience working with OutSystems.

## 5.3 Execution

The execution of this study was done in a period of three weeks. There were two groups of five participants each, because five of them work at Braga's OutSystems office, whereas the remaining five, work at Linda-a-Velha's office, the place where we conducted the study. To conduct the study with the remote participants in Braga, we meet using a video conferencing tool called Zoom. Zoom allowed us to be connected even at a distance, and by using the sharing screen and camera facilities of the tool, we could fully see the other person's actions throughout the study.

Before the start of each study, the GitHub repository that participants would be working with was created with an already built application (OML) and some commit history to simulate a real scenario.

The study was carried individually with each participant and set to take a maximum of thirty minutes with each participant. Initially, we started by introducing the objectives of this thesis to give context to the participant. Next, a brief explanation of the proof of concept produced - OSGit - was given with a small demo of the navigation in the tool while showcasing its functionalities. The following step was to ask the participant to connect to the virtual machine that held the Service Studio version with the OSGit integration and the exercise sheet shown in appendix B. Before the participants started reading and doing the tasks depicted in the exercise sheet, we asked them to think aloud, so that we could understand their line of thought. The participant then started to read the exercise sheet, and began to complete the required tasks. After completing the second task in the "Onto the Exercise" section of the sheet, we also partook in the task, by committing a previously prepared version of the same OML the user was working on, so that when the participant completed task 4, conflicts would appear and need to be solved by the participant accordingly. Lastly, at the end of the exercise, the participant was asked to fill the post-test questionnaire shown in appendix A, and if there was any time left, he/she was free to give any further comments. At the end of each session, the answers to the post-test questionnaire (Appendix A), the end-session feedback and any comments made by the participants throughout the study, were all collected for further analysis.

## 5.4 Results

After completing the usability tests with the ten participants, the results present in the answers to the post-test questionnaire in appendix A were gathered. Appendix C shows the distribution of the answers to the questions one to seven of the "About our Tool" section of the questionnaire. The answers are given in a numerical agreement scale, where the number one represents the "strongly disagree" answer and number five the "strongly agree" answer.

| Question | Min | Max | Median | Mean |
|---|---|---|---|---|
| 1. The interface of our tool, is easy to use. | 4 | 5 | 4 | 4.4 |
| 2. The Settings feature, is easy to use. | 3 | 5 | 4.5 | 4.2 |
| 3. The Commit feature, is easy to use. | 4 | 5 | 4 | 4.3 |
| 4. The Blame feature, is easy to use. | 3 | 5 | 4 | 4.2 |
| 5. The Commit feature would be useful to incorporate with your current software development methodologies, while using OutSystems. | 4 | 5 | 5 | 4.6 |
| 6. The Blame feature would be useful to incorporate with your current software development methodologies, while using OutSystems. | 3 | 5 | 5 | 4.6 |
| 7. Our tool would be useful to incorporate with your current software development methodologies, while using OutSystems. | 4 | 5 | 5 | 4.7 |

Table 5.2: The mean of the answers the first seven questions of the "About our Tool" section of the post-test questionnaire (Appendix A).

Table 5.2 shows the mean answers of the answers displayed in appendix C. These means represent each answer's mean value of the ten given answers for each question. Furthermore, these seven questions are meant to find and establish a level of necessity and usefulness for OSGit.

| Question 8: What difficulties did you have while using our tool? | |
|---|---|
| Participant Number | Answer |
| 1 | "n/a" |
| 2 | "Configuring the settings, was not clear what to define at the beginning, due to the instructions provided in the paper." |
| 3 | "When using the blame it should open directly on the action we've selected in order to compare with the version commited" |
| 4 | "None" |
| 5 | "Linking accounts" |
| 6 | "I can only commit a single OML at a time. I would like to commit the entire feature, which can include multiple OMLs and Appplications." |
| 7 | "Nothing except the double-click thing" |
| 8 | "To be able to commit changes inside a screen or an action (OutSystems does not allow it)" |
| 9 | "No major difficulties, just minor usability issues that can be improved." |

Table 5.3: Participants' answers to question eight of the post-test questionnaire.

For the remaining three answers in the "About our Tool" section of the questionnaire, as they were open-ended questions, the participants had the freedom to express themselves through a small text. Table 5.3 shows the answers to the eighth question regarding the difficulties the participants had while using OSGit. The participants answers will allow us to understand the faced difficulties while using our tool and allow us to refine OSGit so that future problems can be solved.

| Question 9: Which other new features would you like to see in our tool? | |
|---|---|
| Participant Number | Answer |
| 1 | "On ClickPublish be able to Commit." |
| 2 | "Branches and History. Open in Git button, to show how the code repository looks like on Git." |
| 3 | "Have a feature to tag stable versions in order to revert to them if necessary" |
| 4 | "All" |
| 5 | "I would like to see the blame feature with more details. For instance, I want to perform blame in an aggregate inside an action, or in an entity attribute." |
| 6 | "I would love to have the Blame functionality working with the standard OS version control" |
| 7 | "Ability to integrate with other source control systems; branching" |

Table 5.4: Participants' answers to question nine of the post-test questionnaire.

In table 5.4, the participants present their ideas for future features that they would like to see implemented in OSGit. This question is rather interesting as it shows us the various necessities of each area of expertise within OutSystems, because participants come from different teams.

Lastly, table 5.5 shows the suggestions about the general improvements to OSGit that were given by the participants. As it can be observed, the last answer suggesting a better merge operation slides way from the goal of this thesis. Nevertheless, it still shows the satisfaction levels of each participant with the presented functionalities.

| Question 10: Which improvements would you suggest to our tool? | |
|---|---|
| Participant Number | Answer |
| 1 | “N/a” |
| 2 | “When doing a publish, it should also commit to Git.” |
| 3 | “If possible in the future make the 1 click publish also commit to the versioning tool, and also allow several instances of the blame to opened at the same time without the need to close the one currently open” |
| 4 | “Some ui tunes in the views and the link to the gitub account steps” |
| 5 | “I would love some shortcut keys, I don’t like menus much and much less two-level menus :)” |
| 6 | “The pop up with the commit message could be in the first pop up that is showed to the user.” |
| 7 | “Minor usability improvements:<br>- Integrated login in the Setting screens<br>- When commiting, avoid showing the "There are changes" popup (it is unnecessary and causes confusion)<br>- When commiting with errors on the oml, show an alert saying so.<br>- When commiting with conflicts, the experience can be closer to the "1-click publish"<br>- The Blame popup design can be improved for better usability - it can be bigger and the lines can be clearer they’re clickable and what that does (maybe they should have a button instead)” |
| 8 | “A better merge” |

Table 5.5: Participants’ answers to question ten of the post-test questionnaire.

### 5.4.1 The SUS

The System Usability Scale (SUS) was the last section of the post-testing questionnaire - shown in appendix A - handed to the participants at the end of the exercise. To complement the already discussed questionnaire topics, and obtain a more thoroughly usability survey, the SUS was thus presented.

The SUS [6] is a scale of ten questions that assess key usability points for any given system. Each question's answers are given by a numerical agreement degree, i.e., for each question, there is a one to five numerical agreement scale. This scale contemplates the "strongly disagree" answer represented by number one, throughout the next numbers of the scale until number five, which represents a "strongly agree" answer.

| SUS Question | Mean Answer |
|---|---|
| 1. I think that I would like to use this system frequently. | 4,4 |
| 2. I found the system unnecessarily complex. | 1,6 |
| 3. I thought the system was easy to use. | 4,4 |
| 4. I think that I would need the support of a technical person to be able to use this system. | 1,5 |
| 5. I found the various functions in this system were well integrated. | 3,8 |
| 6. I thought there was too much inconsistency in this system. | 1,6 |
| 7. I would imagine that most people would learn to use this system very quickly. | 4,3 |
| 8. I found the system very cumbersome/awkward to use. | 1,6 |
| 9. I felt very confident using the system. | 4,3 |
| 10. I needed to learn a lot of things before I could get going with this system. | 1,8 |

Table 5.6: The mean of the answers for each of the SUS questions.

Table 5.6 shows the mean values of the agreement scale referred to before (from one to five). The evaluation phase contemplated ten people, and the "Mean Answer" column of the table represents the mean value of all of the ten answers to each question. These values are going to enter in the calculation of the global SUS score [6]. Recurring to the values in table 5.6 the global SUS score obtained is 82,75, which is a promising result.

| Score | Acceptability | Rating |
|---|---|---|
| ≤25 | Not Acceptable | Worst Imaginable |
| 26-39 | Not Acceptable | Poor |
| 40-52 | Not Acceptable | Ok |
| 53-74 | Marginal | Good |
| 75-85 | Acceptable | Excellent |
| 86-100 | Acceptable | Best Imaginable |

Table 5.7: The various SUS scores with their acceptability and ratings [4]

## 5.5 Analysis

The results obtained in the study are very promising. By looking at table 5.2, the answers are very positive. These questions focused on the participants global difficulties of using OSGit and if they find the need for incorporating our tool in their software development workflow. The answers means were well above four, which means that it tends to the "strongly agree" side of the statements asked. This means that the participants found few setbacks while using OSGit, and would like to use it as a part of their daily routines.

Regarding the difficulties encountered and shown in table 5.3, the main aspects pointed out were in configuring the GitHub account in the settings screen: as pointed in the second and fifth answer; and minor usability issues: pointed in the third, seventh, eighth and ninth answers. These aspects will be taken into consideration for the future work on this tool, as the difficulties the participants had are of utmost importance.

Table 5.4 displays additional features that the participants suggested to be integrated into OSGit. This also reflects the needs that the developers from different teams have.

The first suggestion pointed by one of the participants is for the 1-Click-Publish OutSystems functionality to also perform a commit. This functionality deploys a running version of the application he/she is producing and stores that version of the application in the OutSystems repository. The 1-Click-Publish functionality has a similar workflow as the commit operation, and currently, it is how the users are used to store and generate versions of their applications. This similarity, lead to the suggestion for the 1-Click-Publish to integrate the commit functionality, thus enabling the commit operation and the OutSystems publish operation to be concentrated in only one button, that the current OutSystems users already use. However, the implications of this integration should be discussed further and the impact of its usability by the users should be measured accordingly, before taking any decisions.

The second subject states that branches and history would be two exciting features to add to our solution. Taking into consideration the branching setback described in section 3.7.2, it would be an interesting feature to incorporate once the database architecture issue is tackled. The history operation to see the full commit history of the repository is also suggested. Lastly, the possibility to integrate the remote repository content visualization through the click of a button is presented by the participant.

Answer five is an interesting observation. The participant suggests that the blame operation to be more detailed, i.e., to be able to show the modification history for items with a finer granularity. For example, an action is constituted by a set of finer granularity elements which are action nodes as depicted in figure 5.1. The suggestion is enabling the blame functionality in OSGit to also work at a deeper level to be able to extract information about each node modifications.

The seventh subject's answer mentions the integrations with other SCS besides Git. Also, the branching operation is once again referred. This seems to indicate this is one of the most wanted features.

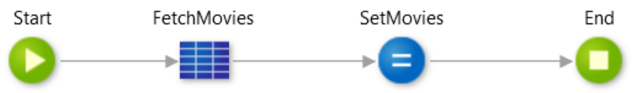


Figure 5.1: An example of an Action content. These four elements that constitute this action are called action nodes.

Regarding the last question of the “About our Tool” section shown in table 5.5, it focuses on the suggested improvements for our tool. As we can see the topic of integrating the 1-Click-Publish OutSystems functionality with the OSGit commit operation appears again in answers two and three, thus revealing the importance of this topic. The remaining suggestions refer UI improvements such as the five minor usability improvements in question seven, the order of the feedback and input messages screen in question six, and allow for several blame screen to be opened, as requested in question three. The last answer targets a topic that is outside the scope of this thesis, which is the improvement of the current merge algorithm provided by OutSystems.

To conclude, we obtained great feedback from the evaluation phase, and the participants showed great satisfaction towards OSGit. Which by analyzing table 5.7 and from the SUS score obtained, with a value of 82.75, we can say that the result obtained for OSGits is acceptable and rated “excellent”. Despite obtaining an acceptable solution, there is still room for improvements according to the participants’ feedback seen in tables 5.3, 5.4 and 5.5.

# Chapter 6

# Conclusions

In this chapter, we present the final observations and the future work for this thesis.

## 6.1 Final Observations

The growing world of VPLs and consequently low-code platforms will always bring the need to incorporate version control methodologies in the software development processes, as teams get bigger within a collaborative environment to accelerate the development process.

The main problem we studied is the integration of a text-based SCS with a VPLs, which answers the following question: how to cope with the loss of SCS functionalities when dealing with visual artifacts? A solution to this problem is what the implemented proof of concept demonstrates. In the context of this thesis, we are integrating Git - the SCS - with OutSystems - the VPL - more precisely with the OutSystems developing environment, the Service Studio. In our solution, the Git clone, commit, push and pull operations, could be directly used with the OutSystems application files, the OMLs. However, the Git compare, merge, and blame operations could not be directly used with the OML files, because these last three operations need to manage or display the file contents, and text-oriented SCS interpret visual assets as black-boxed constructs. So, to tackle the compare and merge operation setback, OutSystems already provides such facility. For the compare and merge cases, instead of using Git's operations, we had to reuse the one that was already built by OutSystems and was ready to interpret and modify the OML files. The blame operation was a different case. The solution for this issue had to be entirely engineered through the creation of metadata files, that by containing text, enabled the Git blame operation to work over them as explained in section 4.5.

With the produced proof of concept that provided the commit and blame operations to be used on OutSystems visual binary files, the OMLs, we proved that integrating a text-based SCS with a VPL is possible. However, issues while integrating operations that manage the file contents will arise when used on visual artifacts. The issues must then be solved accordingly, which leads us to the next concluding point.

Developing these types of solutions to get around the SCS operations loss while handling visual artifacts will be very attached to the technology where these solutions are being developed. For instance, in our solution, the contents of the metadata files are metadata information that represents application building elements provided by the Service Studio. However, if we had another technology that differs from OutSystems, the metadata solution could be different, and perhaps the extracted metadata content may be different due to the nature of the structure of the visual artifacts. The key aspect here is that visual artifacts are vast, and different artifact structures require different solutions. One last example is the compare and merge feature, shown in figure 3.1. This feature compares and merges OMLs files and it is capable to do so, by having a specific algorithm that has the knowledge of the inner file structure of OMLs. Moreover, extracting metadata from the visual elements as way to assist some text-based SCS operations, revealed to be a great solution to tackle this problem and similar ones.

Regarding the participants' feedback obtained in the evaluation phase, described in chapter 5, the results were very positive. The operations provided by the implemented solution caused a good impact on the developing workflow. Mainly the blame operation was the promising one as shown in the results obtained, and throughout comments made by the participants that showed a high level of satisfaction for this operation. However, by looking at the difficulties encountered and the improvements suggested by the participants, we can see that there is still space for improving our tool, mainly by adding more features that would enrich our solution even further.

## 6.2 Future Work

There are still some important aspects that remain to be approached. An introduction to these aspects will be presented next, as the future work for this thesis:

- Working with multiple OMLs: The current solution is only capable of handling one OML per repository. One of the participants in the evaluation phase in table 5.3, pointed out that it is not the optimal solution as many applications produced with OutSystems have two or more OMLs. In the current solution, only the commit operation would need to be targeted and modified in order to work with more than one OML. The blame and settings functionalities are already prepared to handle multiple OMLs in the same repository.

- More repository hosts: Our solution only uses GitHub as the repository host, as the SCS used is Git. However, other repository hosts could be available: like GitLab or BitBucket, for example. This would provide the user with an extended range of available repository hosts, instead of forcing them only to use GitHub. It would be necessary to modify the settings screen of OSGit, by allowing the users to choose among the available repository hosts, and provide the necessary authentication process according to the repository host that was selected.

- More SCS: The integration made with the Service Studio was with Git. Nevertheless, more SCS could also be interesting to make available for the users, as some of them might be keener or used to the workflow of a different SCS, like SVN or Mercurial, for example. For this, we just needed to change the library that establishes the communications with Git and the used Git commands.

- Implement our solution as a plugin: OutSystems allows the creation of plugins inside the platform that provide a given functionality. The difference of a particular functionality being developed as a plugin is that the source code is not inside the source code of the OutSystems product. So, by having our solution implemented as an OutSystems plugin, we could have various pieces of the same solution, that only differed in the SCS chosen to integrate with the platform. For example, there could be a plugin for Git, Mercurial, and SVN. The user can choose the SCS he/she wants to use and install the respective plugin. Thus, his preferred SCS integration is ready to be used, and the integration of various SCS with the platform is easier to manage. To enable OSGit as an OutSystems plugin, it would be necessary to enable certain OutSystems APIs so that the communications with the OutSystems components - mainly the Service Studio and the Platform Server - could be performed.

- The 1-Click-Publish issue: The 1-Click-Publish, as described in section 5.5, has the possibility of being integrated with the commit operation. This must be investigated further as the commit operation provided, also stores a version of the application in a GitHub repository. This may initially cause some confusion, as two repositories are holding the same versions of the user's applications, where its operations need to be called from two different places. So, ideally, the commit and 1-Click-Publish should be investigated, and an eventual merge of these two operations should also be considered, as suggested by two participants from the validation phase in table 5.5.

# Bibliography

[1] *About OutSystems - Facts and Figures | OutSystems*. URL: https://www.outsystems.com/company/ (visited on 02/01/2019).

[2] *AMOR - Adaptable Model Versioning*. URL: http://www.modelversioning.org/ (visited on 01/08/2019).

[3] *Apache Subversion*. URL: https://subversion.apache.org/ (visited on 02/14/2019).

[4] A. Bangor, P. Kortum, and J. Miller. "Determining what individual SUS scores mean: Adding an adjective rating scale." In: *Journal of usability studies* 4.3 (2009), pp. 114–123.

[5] *Bazaar*. URL: http://bazaar.canonical.com/en/ (visited on 02/14/2019).

[6] J. Brooke et al. "SUS - A quick and dirty usability scale." In: *Usability evaluation in industry* 189.194 (1996), pp. 4–7.

[7] *CDO - The model repository*. URL: https://www.eclipse.org/cdo/ (visited on 01/08/2019).

[8] *CDO Model Repository Overview*. URL: https://www.eclipse.org/cdo/documentation/ (visited on 09/23/2019).

[9] *Compare and merge example with conflicts - OutSystems*. URL: https://success.outsystems.com/Documentation/11/Developing_an_Application/Merge_the_Work/Compare_and_merge_example_with_conflicts (visited on 02/07/2019).

[10] *Compare Repositories - Open Hub*. URL: https://www.openhub.net/repositories/compare (visited on 07/20/2019).

[11] *CVS - Open Source Version Control*. URL: https://www.nongnu.org/cvs/ (visited on 02/14/2019).

[12] *CVS — Concurrent Versions System v1.11.23l*. URL: https://www.gnu.org/software/trans-coord/manual/cvs/cvs.html (visited on 02/14/2019).

[13] J. S. Dumas, J. S. Dumas, and J. Redish. *A practical guide to usability testing*. Intellect books, 1999, pp. 4–6.

[14] *Eclipse Subversive - Subversion (SVN) Team Provider*. URL: https://www.eclipse.org/subversive/ (visited on 02/19/2019).

[15] *EMF Compare - Compare and Merge Your EMF Models.* URL: https://www.eclipse.org/emf/compare/overview.html (visited on 01/08/2019).

[16] *Git.* URL: https://git-scm.com/ (visited on 02/14/2019).

[17] *Git - git-log Documentation.* URL: https://git-scm.com/docs/git-log (visited on 09/22/2019).

[18] *Git Credential Manager for Windows.* URL: https://github.com/microsoft/Git-Credential-Manager-for-Windows (visited on 09/02/2019).

[19] *GitHub for Unity.* URL: https://assetstore.unity.com/packages/tools/version-control/github-for-unity-118069 (visited on 01/16/2019).

[20] *GNU CSSC.* URL: https://www.gnu.org/software/cssc/ (visited on 02/07/2019).

[21] *Helix - Triggers.* URL: https://www.perforce.com/perforce/doc.current/manuals/p4sag/Content/P4SAG/chapter.scripting.html (visited on 02/20/2019).

[22] *Helix Plugin for Unity (P4Connect).* URL: https://www.perforce.com/plugins-integrations/p4connect-unity-plugin (visited on 02/20/2019).

[23] *Helix Plugin for Visual Studio Source Control.* URL: https://www.perforce.com/plugins-integrations/visual-studio-plugin (visited on 02/20/2019).

[24] *Helix Plugins Integrations.* URL: https://www.perforce.com/plugins-integrations (visited on 02/19/2019).

[25] *Hg - Visual Studio Marketplace.* URL: https://marketplace.visualstudio.com/items?itemName=mrcrowl.hg (visited on 02/19/2019).

[26] *How does OutSystems enable team collaboration? - OutSystems.* URL: https://success.outsystems.com/Evaluation/Lifecycle_Management/How_does_OutSystems_enable_team_collaboration (visited on 02/05/2019).

[27] A. Koc and A. Tansel. "A Survey of Version Control Systems." In: (Feb. 2019).

[28] M. Koegel and J. Helming. "EMFStore: a model repository for EMF models." In: *2010 ACM/IEEE 32nd International Conference on Software Engineering.* Vol. 2. 2010, pp. 307–308. DOI: 10.1145/1810295.1810364.

[29] *LibGit2Sharp.* URL: https://github.com/libgit2/libgit2sharp (visited on 08/30/2019).

[30] H. Lourenço and R. Eugénio. "TrueChange™ under the hood: how we check the consistency of large models (almost) instantly." In: (2019).

[31] *Low-Code Adoption Is on the Rise: Key Findings from the State of Application Development Report.* URL: https://www.outsystems.com/blog/posts/low-code-adoption/ (visited on 09/20/2019).

[32] J. N. Macedo, R. Moreira, J. Cunha, and J. Saraiva. "Get Your Spreadsheets Under (Version) Control." In: *Proceedings of XXII Ibero-American Conference on Software Engineering (CIbSE 2019), Software Engineering Track.* to appear. Apr. 2019.

[33] *Mendix: We Help Enterprises Achieve their Digital Goals with Low-code.* URL: https://www.mendix.com/company (visited on 02/20/2019).

[34] *Mercurial SCM.* URL: https://www.mercurial-scm.org/ (visited on 02/14/2019).

[35] *MercurialEclipse.* URL: https://marketplace.eclipse.org/content/mercurialeclipse (visited on 02/19/2019).

[36] *ModelBus Overview.* URL: https://www.modelbus.org/en/modelbusoverview.html (visited on 01/08/2019).

[37] R. M.M. K. Moreira. "SheetGit: A Tool for Collaborative Spreadsheet Development." Master's thesis. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2016.

[38] *OutSystems tools and components.* URL: https://success.outsystems.com/Evaluation/Architecture/1_OutSystems_Platform_tools_and_components#Development_environments (visited on 02/04/2019).

[39] *Perforce - Documentation.* URL: https://www.perforce.com/support/self-service-resources/documentation (visited on 02/19/2019).

[40] *Perforce | HelixCore - Version Control Software for Accelerated Development.* URL: https://www.perforce.com/products/helix-core (visited on 02/14/2019).

[41] *Perforce Open Sources Popular Version Control Tools | Perforce.* URL: https://www.perforce.com/press-releases/perforce-open-sources-popular-version-control-tools (visited on 02/22/2019).

[42] *Plastic SCM - The Distributed Version Control for Big Projects.* URL: https://www.plasticscm.com/features (visited on 02/14/2019).

[43] *Plastic SCM - Unity plugin.* URL: https://www.plasticscm.com/unity-plugin (visited on 02/19/2019).

[44] *Plastic SCM version control in Microsoft Office.* URL: https://www.plasticscm.com/documentation/office/plastic-scm-version-control-office-guide (visited on 02/19/2019).

[45] M. Revell. *What Is Low-Code?* URL: https://www.outsystems.com/blog/what-is-low-code.html (visited on 02/01/2019).

[46] *Source Control.* URL: https://docs.unrealengine.com/en-US/Engine/UI/SourceControl (visited on 01/16/2019).

[47] *Source Control Inside Unreal Editor.* URL: https://docs.unrealengine.com/en-US/Engine/Basics/SourceControl/InEditor (visited on 01/16/2019).

[48] *Subversion - Help | IntelliJ IDEA.* URL: https://www.jetbrains.com/help/idea/using-subversion-integration.html (visited on 02/19/2019).

[49] *The Mendix Difference for Platform as a Service Providers.* URL: https://www.mendix.com/the-mendix-difference/ (visited on 02/20/2019).

[50] W. F. Tichy. "RCS—a system for version control." In: *Software: Practice and Experience* 15.7 (1985), pp. 637–654.

[51] *Unity - Manual: Version control integration*. URL: https://docs.unity3d.com/Manual/Versioncontrolintegration.html (visited on 01/16/2019).

[52] *Using Perforce as Source Control*. URL: https://docs.unrealengine.com/en-us/Engine/Basics/SourceControl/Perforce (visited on 01/16/2019).

[53] *Version and source control - OutSystems*. URL: https://success.outsystems.com/Evaluation/Lifecycle_Management/Version_and_source_control (visited on 02/04/2019).

[54] *Version Control - Mendix 7 Reference Guide | Mendix Documentation*. URL: https://docs.mendix.com/refguide/version-control (visited on 02/20/2019).

[55] *Version Control in Visual Studio Code*. URL: https://code.visualstudio.com/docs/editor/versioncontrol (visited on 02/19/2019).

[56] *Version Control Management Tools  Multi-user Application Development | Mendix Evaluation Guide*. URL: https://www.mendix.com/evaluation-guide/app-lifecycle/version-control (visited on 02/20/2019).

[57] *VS annotate or Git's blame*. URL: https://www.outsystems.com/ideas/5351/VS+annotate+or+Git's+blame?IsFromAdvancedSearch=True#IdeaComment19941 (visited on 02/21/2019).

[58] *What is EMFStore and why should I use it?* URL: https://www.eclipse.org/emfstore/ (visited on 01/08/2019).

[59] *What is Source Control? - Amazon Web Services*. URL: https://aws.amazon.com/devops/source-control/ (visited on 02/01/2019).

Appendix A

# Post-Test Questionnaire

The post-testing questionnaire presented to the testers after completing the testing exercise shown in appendix B.

## User Acceptance Questionnaire

* Required

1. **Gender** *

   *Mark only one oval.*

   ◯ Male

   ◯ Female

2. **Age** *

   *Mark only one oval.*

   ◯ < 20

   ◯ 20-30

   ◯ 30-40

   ◯ 40-50

   ◯ > 50

3. **What is your education level?** *

   ______________________________

4. **What is your major?**

   ______________________________

5. **For how long have you been using OutSystems?** *

   ______________________________

## About our Tool

6. **The interface of our tool, is easy to use.** *

   *Mark only one oval.*

|  | 1 | 2 | 3 | 4 | 5 |  |
|---|---|---|---|---|---|---|
| Strongly Disagree | ◯ | ◯ | ◯ | ◯ | ◯ | Strongly Agree |

7. **The Settings feature, is easy to use.** *

   *Mark only one oval.*

|  | 1 | 2 | 3 | 4 | 5 |  |
|---|---|---|---|---|---|---|
| Strongly Disagree | ◯ | ◯ | ◯ | ◯ | ◯ | Strongly Agree |

8. **The Commit feature, is easy to use.** *
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ◯ | ◯ | ◯ | ◯ | ◯ | Strongly Agree |

9. **The Blame feature, is easy to use.** *
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ◯ | ◯ | ◯ | ◯ | ◯ | Strongly Agree |

10. **The Commit feature would be useful to incorporate with your current software development methodologies, while using OutSystems.** *
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ◯ | ◯ | ◯ | ◯ | ◯ | Strongly Agree |

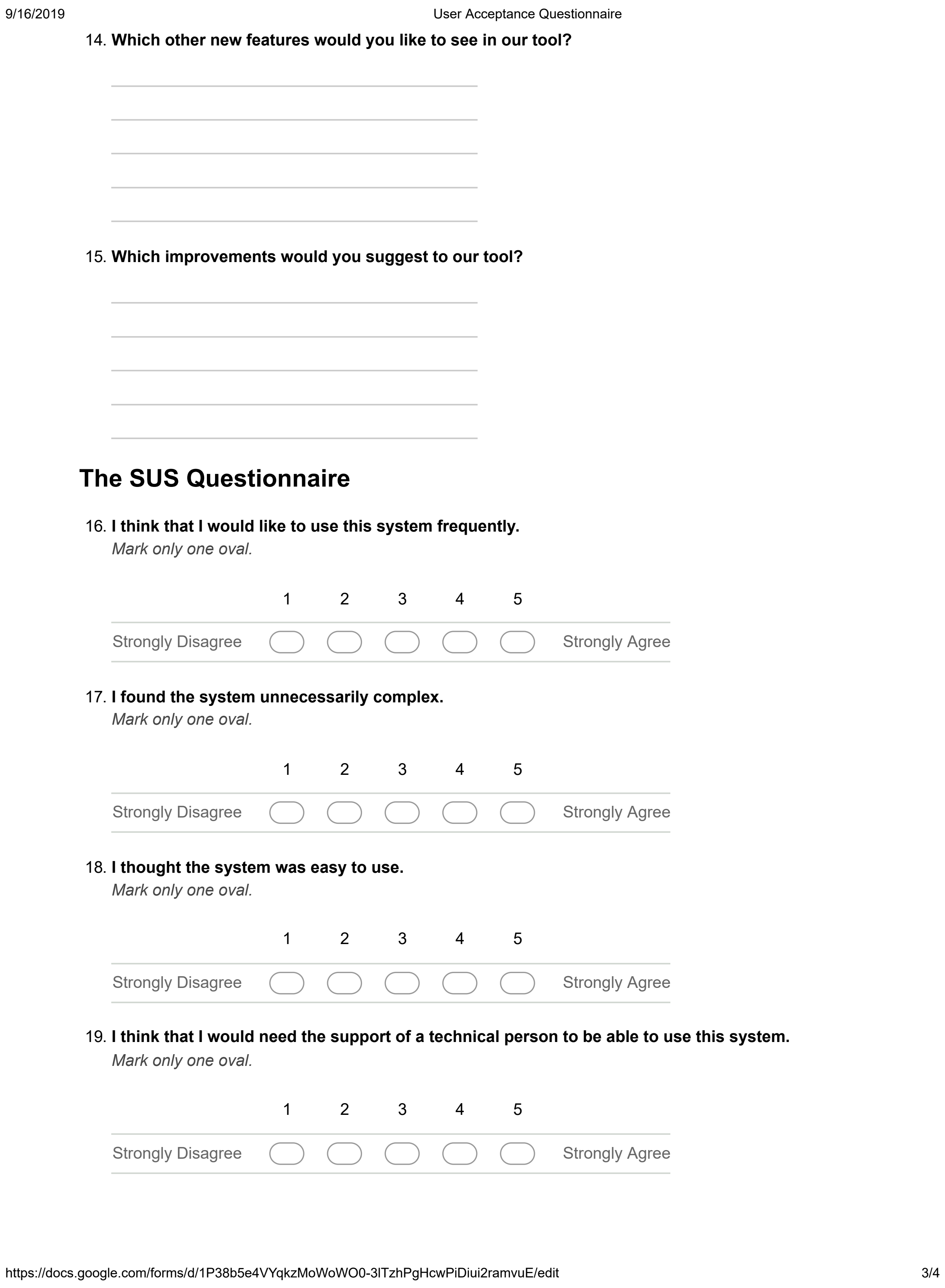
11. **The Blame feature would be useful to incorporate with your current software development methodologies, while using OutSystems.** *
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ◯ | ◯ | ◯ | ◯ | ◯ | Strongly Agree |

12. **Our tool would be useful to incorporate with your current software development methodologies, while using OutSystems.** *
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ◯ | ◯ | ◯ | ◯ | ◯ | Strongly Agree |

13. **What difficulties did you have while using our tool?**



9/16/2019 User Acceptance Questionnaire

14. **Which other new features would you like to see in our tool?**

15. **Which improvements would you suggest to our tool?**

## The SUS Questionnaire

16. **I think that I would like to use this system frequently.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ○ | ○ | ○ | ○ | ○ | Strongly Agree |

17. **I found the system unnecessarily complex.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ○ | ○ | ○ | ○ | ○ | Strongly Agree |

18. **I thought the system was easy to use.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ○ | ○ | ○ | ○ | ○ | Strongly Agree |

19. **I think that I would need the support of a technical person to be able to use this system.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ○ | ○ | ○ | ○ | ○ | Strongly Agree |

https://docs.google.com/forms/d/1P38b5e4VYqkzMoWoWO0-3lTzhPgHcwPiDiui2ramvuE/edit 3/4

20. **I found the various functions in this system were well integrated.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Strongly Agree |

21. **I thought there was too much inconsistency in this system.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Strongly Agree |

22. **I would imagine that most people would learn to use this system very quickly.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Strongly Agree |

23. **I found the system very cumbersome/awkward to use.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Strongly Agree |

24. **I felt very confident using the system.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Strongly Agree |

25. **I needed to learn a lot of things before I could get going with this system.**
*Mark only one oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Strongly Disagree | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Strongly Agree |

# Appendix B

# SCS User Acceptance Exercise

## Setup Phase

Now you will be asked to perform a small exercise, in which you will have to use our tool. In order to do this task, a previously created GitHub repository and a simple OutSystems application will be provided, alongside an existing GitHub account. Usually, the creation of this repository would have to be done by the user, but due to time constraints, the GitHub account, the GitHub repository and the OutSystems application are already provided. You must configure your version control profile, with the data provided below:

- Login to GitHub with these credentials:
  - ☐ Username: ut1-vcs
  - ☐ Password: user1-vcs-2019
- Fill the Git configuration window:
  - ☐ Email: daemon@mail.com
- Remote repository: https://github.com/User383/CinemaApp.git
- Local Repository: choose and create a new folder called "CinemaAppRepo"
- Save the settings screen.

## Onto the Exercise

The application used to do this exercise is very simple. You will be modifying a simplified version of one of the OutSystems tutorial applications: the OSMDb or "Cinemas Application". As you are working on a distributed environment, conflicts may occur and you will need to solve them as you see fit.

1. In the “MovieDetail” web screen, create a new button that deletes the current movie;
2. Commit the current changes to the repository;
3. In the “Movies” web screen, create a new table of records that shows every movie that has a GrossTakingsAmount lesser than 100 milions dollars;
4. Commit the current changes to the repository;
5. Verify who was the first person to modify the "Movies"web screen, while checking his name and message. Also you can click on any row of the blame table, to compare or even merge, your current version with any previous versions.

Appendix C

# SCS User Acceptance Exercise

Each participant's answers to the post-test questionnaire, shown in appendix A are show-cased next.

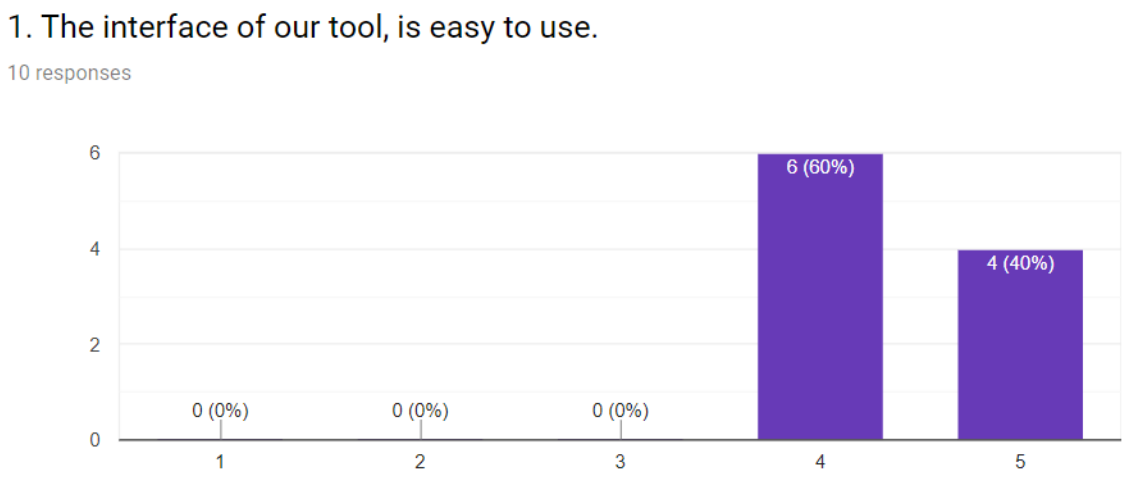


Figure C.1: Participants answers to question one of the post-test questionnaire.

2. The Settings feature, is easy to use.

10 responses

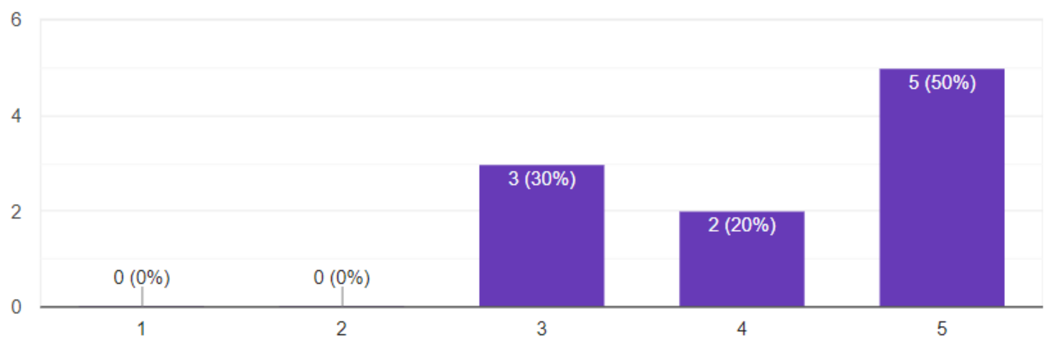


Figure C.2: Participants answers to question two of the post-test questionnaire.

3. The Commit feature, is easy to use.

10 responses

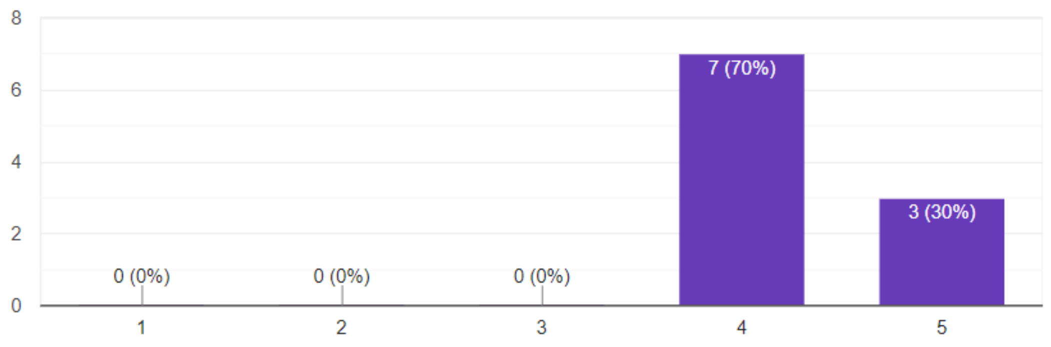


Figure C.3: Participants answers to question three of the post-test questionnaire.

4. The Blame feature, is easy to use.

10 responses

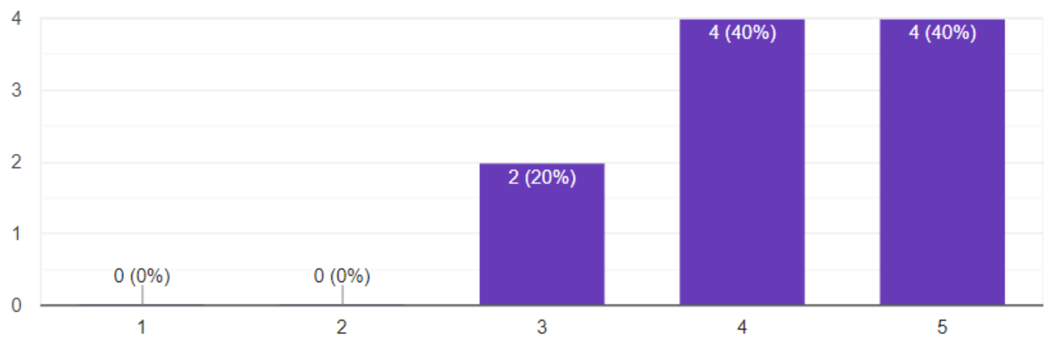


Figure C.4: Participants answers to question four of the post-test questionnaire.

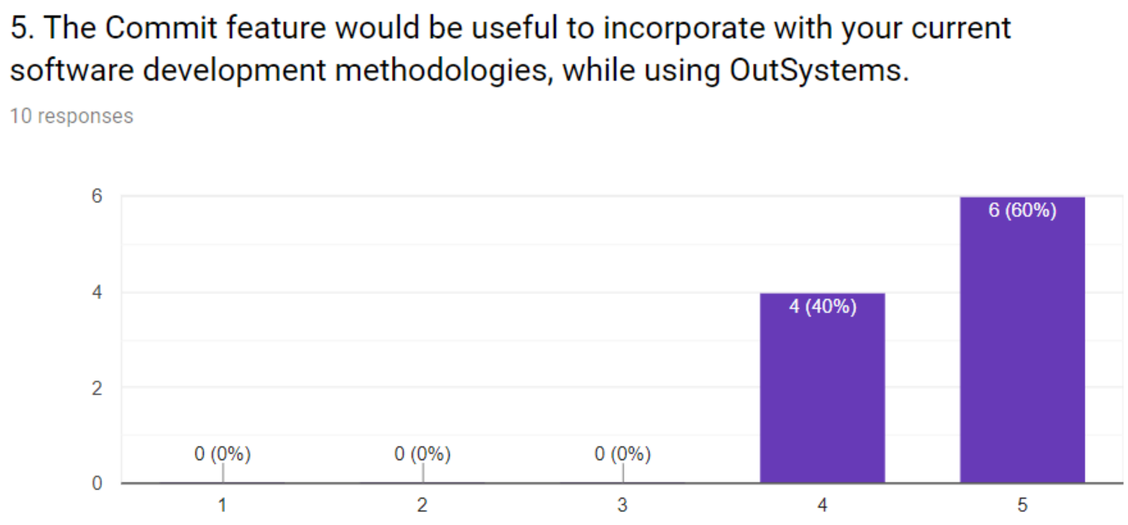


Figure C.5: Participants answers to question five of the post-test questionnaire.

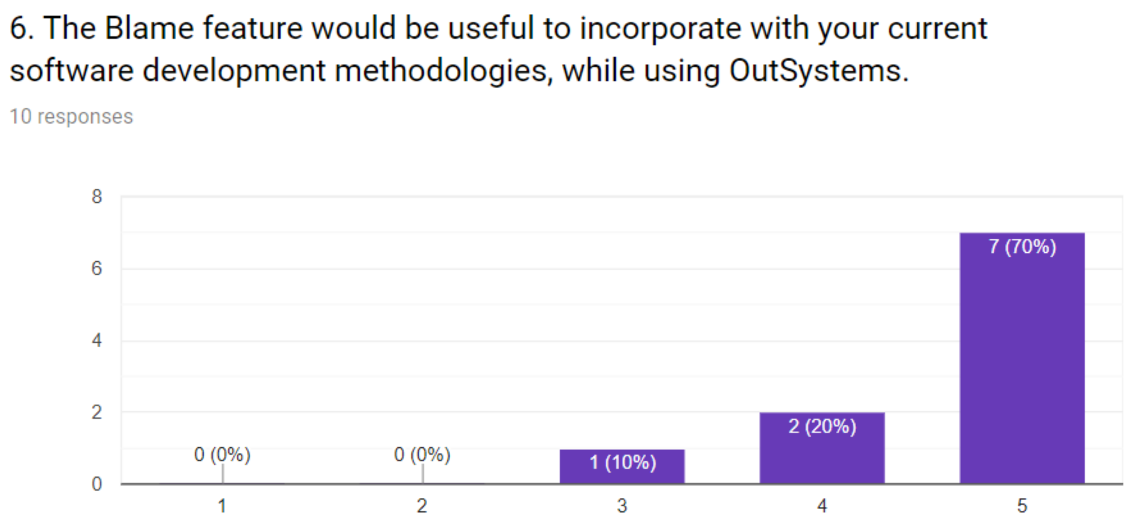


Figure C.6: Participants answers to question six of the post-test questionnaire.

7. Our tool would be useful to incorporate with your current software development methodologies, while using OutSystems.

10 responses

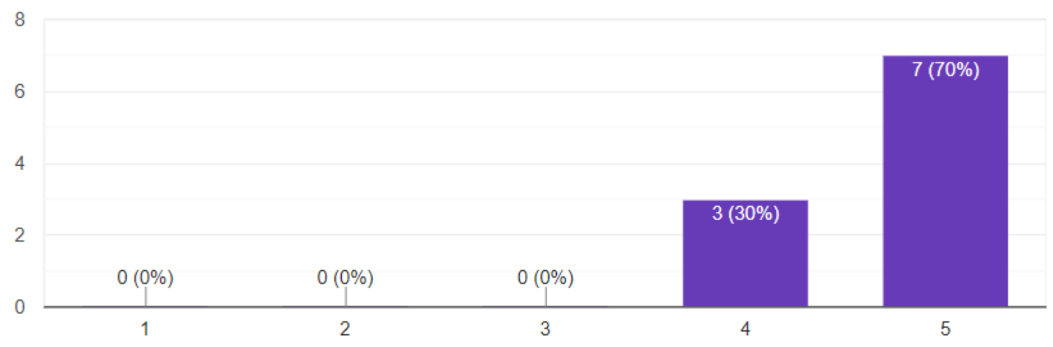
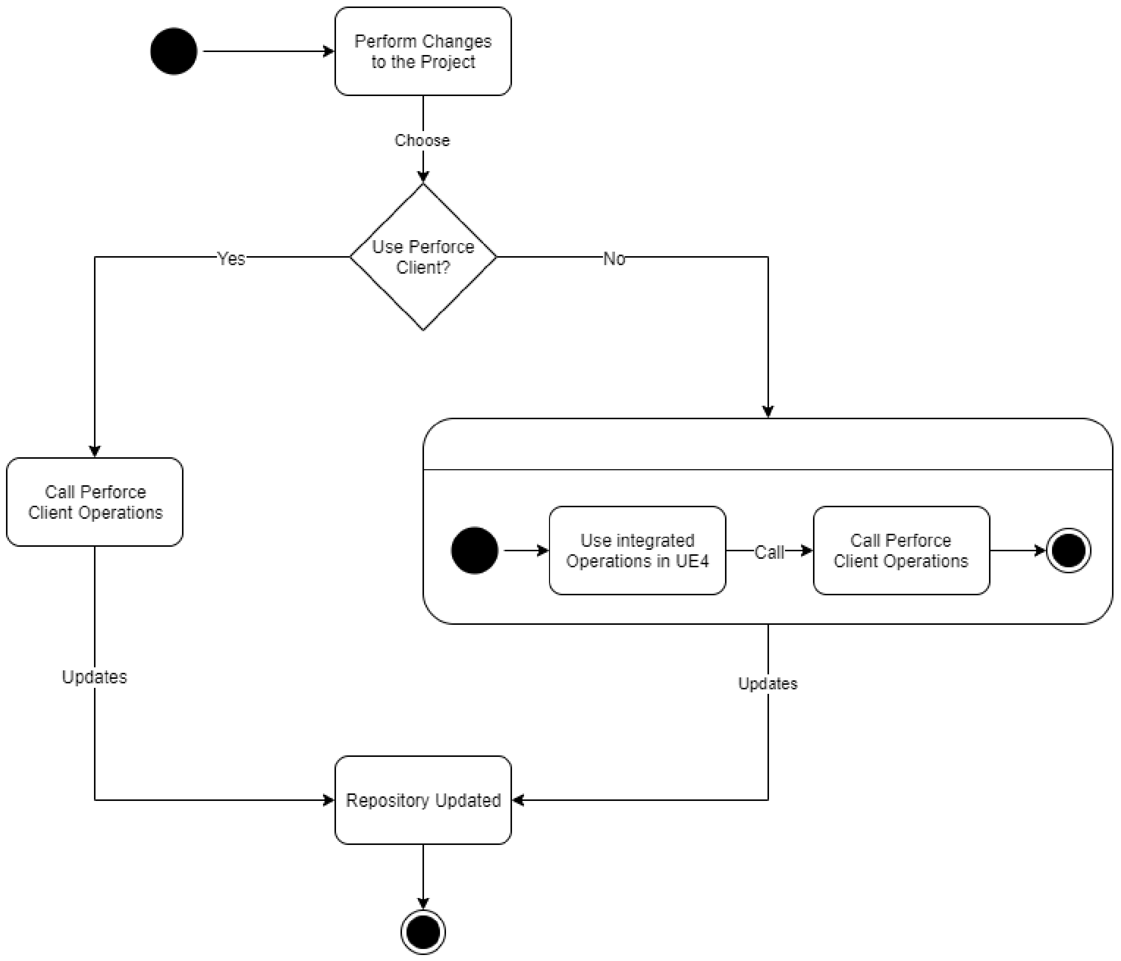


Figure C.7: Participants answers to question seven of the post-test questionnaire.

Appendix D

# SCS workflows in IDE's



Perform Changes to the Project
Choose
Use Perforce Client?
Yes
No
Call Perforce Client Operations
Use integrated Operations in UE4
Call
Call Perforce Client Operations
Updates
Updates
Repository Updated

Figure D.1: State diagram of the source control workflow, while UE4 using Perforce.

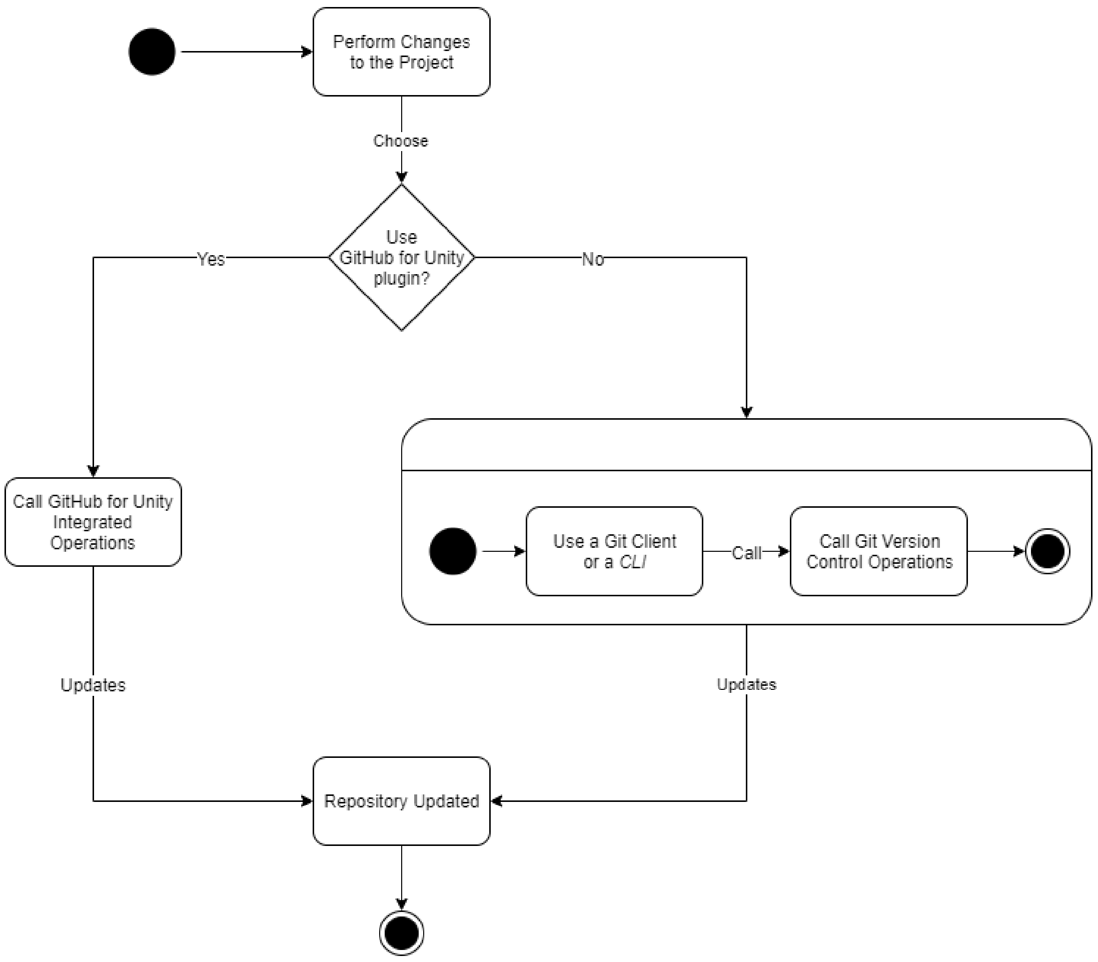

Perform Changes to the Project
Choose
Use GitHub for Unity plugin?
Yes
No
Call GitHub for Unity Integrated Operations
Use a Git Client or a *CLI*
Call
Call Git Version Control Operations
Updates
Updates
Repository Updated

Figure D.2: State diagram of the source control workflow in Unity, while using Git.