Tiago Alexandre Gomes de Almeida

Nº 36656

Real-Time Collaborative Editing of OutSystems DSL Models

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Rodrigo Rodrigues, Associate Professor,
Universidade Nova de Lisboa

Júri:

Presidente: Ana Moreira

Arguente: João Garcia

Vogal: Rodrigo Rodrigues

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2013

Real-Time Collaborative Editing of OutSystems DSL Models

To my family and friends

Acknowledgements

This thesis has been quite a challenge, for different reasons on different times.

I want to thank OutSystems for this opportunity. It was a pleasure working at OutSystems and it was a great opportunity to develop myself. Furthermore, thank you for the internship wage, which allowed me to fully concentrate on the thesis.

I would like to thank my tutor at OutSystems, Hugo Lourenço. Thank you for all the constructive conversations, for the patience to guide me through the OutSystems Platform, for the encouragement and motivation that you gave since the first day at OutSystems.

I thank Lúcio Ferrão, for his insightful ideas. Thank you for the brainstorming sessions we ran and for the ideas that, somehow, came out of that brilliant brain of yours.

I would like to thank my advisor Rodrigo Rodrigues, for his guidance. Thank you for sharing all your knowledge and advisory. I would also like to thank professor Nuno Preguiça, for his wonderful support during the development of this thesis. Thank you, Nuno.

Many thanks to all my colleagues at OutSystems, who received me so well. I would like to especially thank André Simões, Miguel Alves and Nuno Grade, who were the ones I spent my lunch time with. Thank you for your support and shared time.

Finally, I am very grateful for the people around me. My loving mother; my role model, my father; my caring sister; my implicative nephew, who I cannot help but love. My friends with who I party and clear my head. To all you: no words can describe my love and gratitude I feel.

Abstract

Real-time collaborative editing systems are common nowadays, and their advantages are widely recognized. Examples of such systems include Google Docs, ShareLaTeX, among others. This thesis aims to adopt this paradigm in a software development environment. The OutSystems visual language lends itself very appropriate to this kind of collaboration, since the visual code enables a natural flow of knowledge between developers regarding the developed code. Furthermore, communication and coordination are simplified.

This proposal explores the field of collaboration on a very structured and rigid model, where collaboration is made through the copy-modify-merge paradigm, in which a developer gets its own private copy from the shared repository, modifies it in isolation and later uploads his changes to be merged with modifications concurrently produced by other developers. To this end, we designed and implemented an extension to the OutSystems Platform, in order to enable real-time collaborative editing. The solution guarantees consistency among the artefacts distributed across several developers working on the same project.

We believe that it is possible to achieve a much more intense collaboration over the same models with a low negative impact on the individual productivity of each developer.

Keywords: Replicated Systems, Consistency, Collaborative Editing, Collaborative Software Development, Groupware Systems, Change-Based Collaboration

Resumo

Hoje em dia são comuns os sistemas de edição colaborativa em real-time, tais como Google Docs, ShareLaTeX, entre outros. Esta tese visa adoptar este paradigma num ambiente de desenvolvimento de software. A linguagem visual OutSystems é ela própria muito apropriada a este tipo de colaboração, uma vez que, com o código visual, a transferência de conhecimento de código feito entre os programadores torna-se mais fácil. Além disso, simplificam-se a comunicação e coordenação.

Esta tese explora um campo de colaboração sobre um modelo muito estruturado e rígido, em que a colaboração é feita através do paradigma *copy-modify-merge*, no qual os programadores obtêm a sua cópia privada do repositório partilhado, modificam-na isoladamente e, mais tarde, carregam as suas alterações para que se juntem com as modificações concorrentemente produzidas por outros programadores. Para este fim, desenhámos e implementámos uma extensão à OutSystems Platform, por forma a permitir edição colaborativa em tempo real. A solução garante consistência entre os artefatos, distribuídos pelos vários programadores a trabalhar no mesmo projecto.

Acreditamos que é possível ter uma colaboração muito mais intensa sobre os mesmos modelos, com pouco prejuízo para a produtividade individual de cada um.

Palavras-chave: Sistemas Replicas, Consistência, Edição Colaborativa, Desenvolvimento Colaborativo de Software, Sistemas de Trabalho em Grupo, Colaboração Baseada em Alterações

Contents

List of Figures

1

Introduction

Information systems have a major role in today's society. They have become indispensable in many areas, improving process efficiency, information organization, communication, reducing costs, among other contributions.

The time to market for software products is increasingly aggressive and given this demand for development speed, product development needs to become a team effort, instead of a one-man job. Thus, in order to develop a product of higher quality, communication and coordination among developers become fundamental concerns.

Coordinating the efforts of multiple elements of a team working in parallel on the same module is not trivial and a considerable fraction of the effort is spent resolving conflicts, which happen when several people's changes collide in some shared resources. Often, these conflicts are only detected at a late stage when the different updates are merged together.

Real-time collaborative editing systems are an emerging solution to cooperative work as they allow a group of users to view and edit the same document at the same time. Thus, the risk of conflicts is decreased and the group proximity is increased.

1.1 Motivation

In collaborative software development, the project is split into several work items, which are distributed among a set of developers. These work items require the team members to change some common artefacts, such as documents or sections of a document. Finally, when adequate, the changes of the developers are merged. Naturally, a team element's changes to an artefact might have impact on another element's work, when editing related sections of the code.

When the time elapsed between changes and their impact over other team element's changes increases, so does the amount of changes and the risk of conflict when the changes of separate team elements are merged. These conflicts require a developer's attention to be solved. Solving conflicts requires more than a straightforward action from the developers, like inserting the missing code from one version to another. It requires the developer to understand and gain a degree of contextualization for the other developer's code modifications, in order to be able to merge the code modifications of both. These difficulties damage productivity, making the development process inefficient.

A possible approach to avoid such conflicts is to serialize development on specific resources using a "lock" mechanism. Often the elements resort to mechanisms parallel to their working tools, e.g. a physical token whose owner is the only one allowed to perform changes on a specific resource. Although avoiding collisions, these social protocols are impractical on teams whose elements are scattered among distant places, or when communication is hard to maintain. Real-time collaborative editing is in itself a solution for these problems, reducing the risk of conflicts.

Simultaneously, *Google Docs* and other modern *SaaS*[1] applications have raised the bar on the way team members expect to work together. Outdated views of a model are becoming annoyances in the development process. This evolution on collaborative tools drive the providers of software development tools to take a step further and improve their tools, not only to stay competitive, but also to produce attractive solutions for development teams.

[1]Software as a Service (SaaS) is a software delivery model in which software and associated data are centrally hosted on the cloud. SaaS is typically accessed by users via a web browser.

1.2 Context

Projects go through a set of steps during their life cycle. During the planning phase, projects are split into several work items, which are distributed among a set of developers. Due to the advance of collaboration tools, the typical team physical organization is changing, when developers are physically distributed across distant places.

The development process is, in general, performed by multiple developers working on shared artefacts and each developer's changes are likely to be relevant to some other developers.

Efficiency is compromised every time a developer needs to stop working on the product itself and must focus on solving other problems. An interrupted programmer usually takes several minutes to start editing code after resuming work from an interruption [PR11]. Efficiency is damaged not only by wasting time solving undesired issues. When a team needs to solve conflicts and merge works, there is a risk of lost code, due to the lack of context of the people who do the merge.

Collaborative software development is commonly performed in two scenarios:

Scattered usually, when tasks are straightforward, they do not need special attention and can be implemented by a single developer. This scenario involves mostly disjoint work, where each member is focused on a different work item. It is a common scenario when the tasks to be delivered are simple and the requirement is the development speed. In these cases, parallelizing work is the better solution.

Pair-programming this is an agile software development technique in which two developers work together on the same work item. One of them, the driver (or Holmes), writes code while the other, the observer (or Watson), reviews what the driver is coding in. Frequently, the two developers switch roles. While observing, Watson is also able to consider the strategic direction of the work, coming up with ideas for improvements and problems to address. This frees to focus on the "tactical" aspect of completing the task, using the observer as a guide. Pair-programming has proven to result in the production of shorter programs, with better signs and fewer bugs [CW00]. Other non-technical benefits include increased morale [WK03, CW00], knowledge and greater confidence in the correctness of the solution [CW00]. This technique is better suited on risky tasks which require more attention, as it is a

less efficient technique, regarding costs [CW00].

The pair-programming technique is not possible if the team elements are dispersed across distant places. On those situations, a real-time collaborative system would enable another scenario: distributed pair-programming. Distributed pair programming is the use of an agile technique, pair-programming, in a distributed environment, where team elements are dispersed [SW02].

1.3 OutSystems

This thesis is conducted in the context of software development using the OutSystems Platform.

OutSystems is a company that focuses on reducing the costs of custom enterprise software development, using its flagship product, the OutSystems Platform. OutSystems started in 2001 in Portugal. Nowadays, they have two offices in Portugal (Linda-a-Velha and Proença-a-Nova), two offices in the US (Atlanta and San Ramon), in the Netherlands, and also a presence in Brazil and South Africa.

OutSystems has about forty employees working in the R&D team, whose main target is basically evolving the OutSystems Platform, a product for developing enterprise web applications.

1.3.1 OutSystems Platform™

The OutSystems Platform provides support for the full development cycle. Furthermore, the application deployment and evolution cycle is also managed from within the platform. Still, it is an extensible framework allowing for custom built extensions and interconnections with other systems. IT teams around the world use the OutSystems Platform to develop, deploy, manage and change web applications. Applications are often composed by several modules. From now on, we will use the terms “eSpace” to designate a module.

1.3.2 Service Studio

Service Studio is an IDE based on a visual Domain Specific Language (DSL), that covers the definition of business processes, user interfaces, business logic, and data definition and manipulation, web services, security, emails and scheduled jobs. Figure 1.1 shows the Service Studio development environment.

Service Studio is divided in three main components: Content Editor, eSPace Tree and Properties Editor. We now proceed to explain these in more detail.

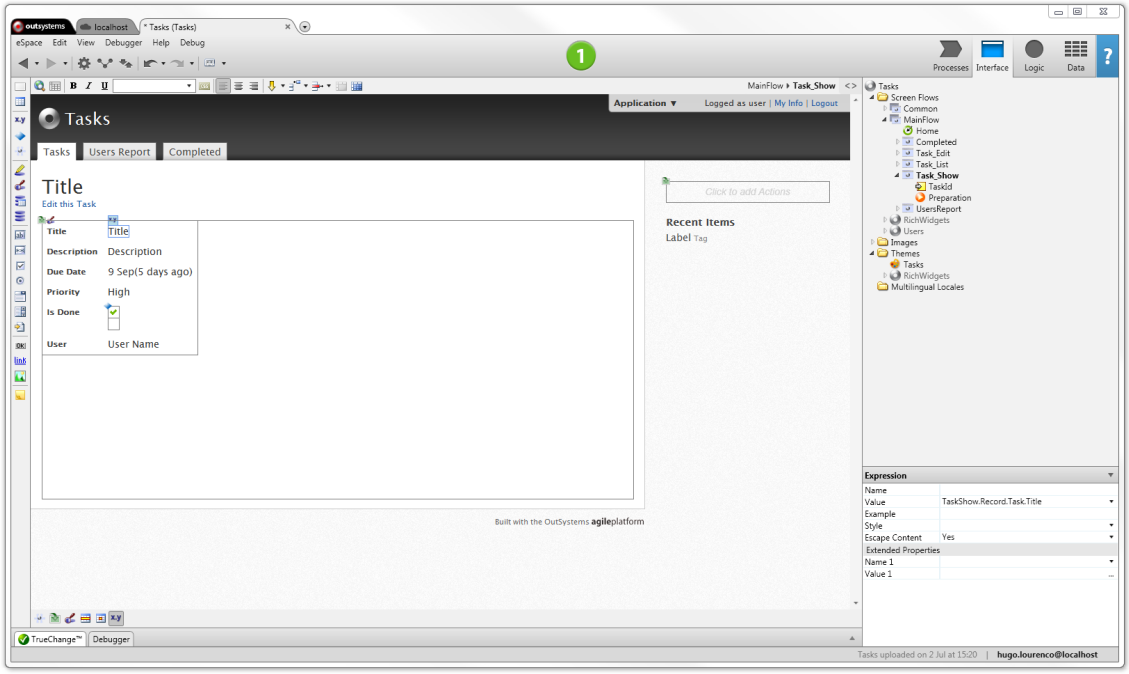


Figure 1.1: Service Studio development environment.

Content Editor

The content editor is the biggest section of the Service Studio. If fills most of the screen. The Content Editor is where developers edit the appearance of a web-page (see Figure 1.2), the behaviour of an action[2], relationships between several database entities, among other activities.

The Figure 1.2 shows the edition of a Web-Screen appearance. The developer needs not to type in any kind of HTML, whatsoever. The creation of a web-screen and edition of its appearance is mostly performed using the developer's mouse, by clicking and dragging objects, such as text boxes, radio boxes, among others.

Furthermore, as mentioned before, the Content Editor is used to edit more than just the appearance of a web-page. A good example is the flow of an action. Figure 1.3 shows the flow editor.

The figure illustrates a flow of an action that checks if the Contacts database table is empty and, if it is, fills the table with information imported from an excel file. As it can be seen through Figure 1.3, developers do not "code in". The development is made through what can be described as an improved flowchart.

[2]an action is the term used to object that, in a regular programming language, would be a function/procedure

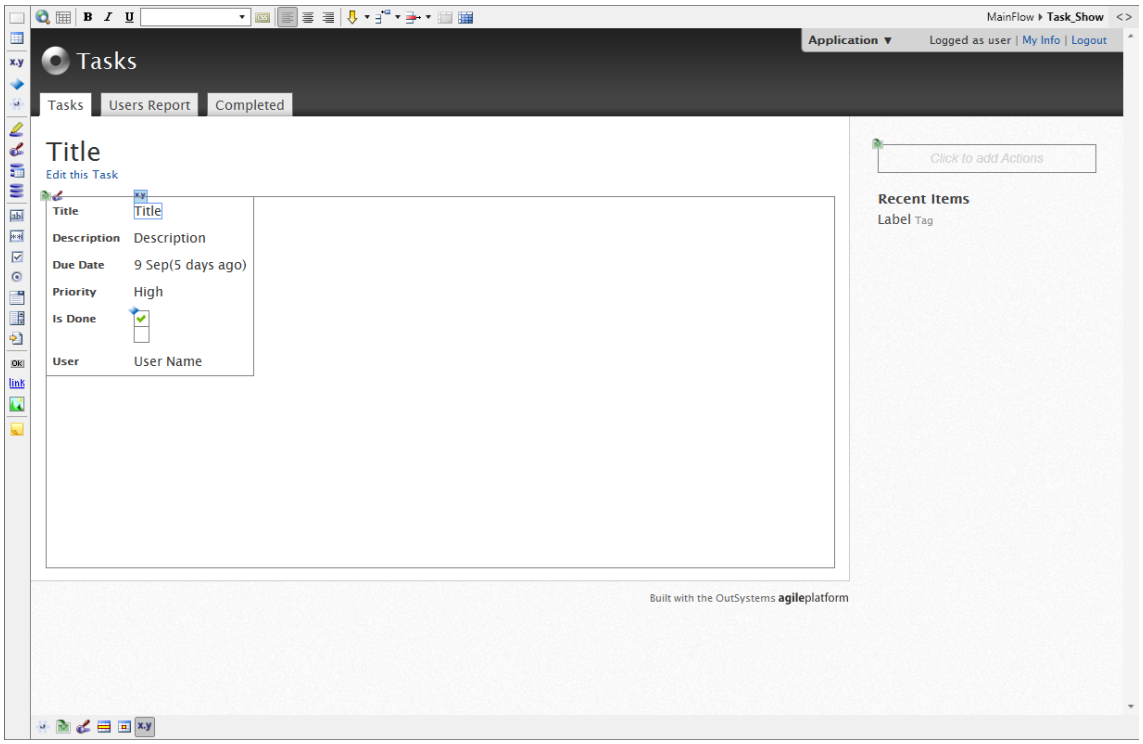


Figure 1.2: Service Studio: Content Editor.

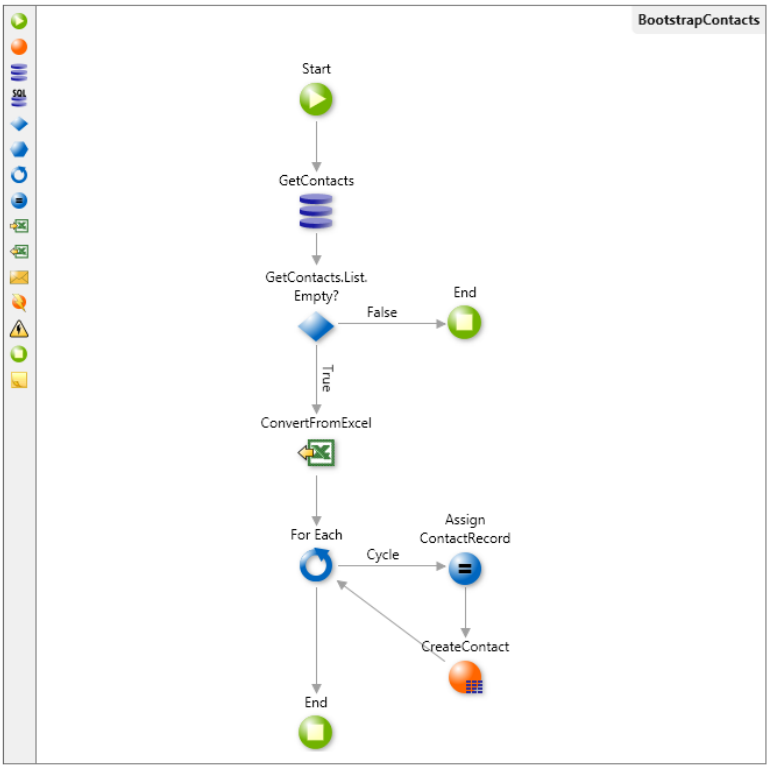


Figure 1.3: Service Studio: Flow Editor.

eSpace Tree

The eSpace Tree is the section that gives a global view of all the application's components, divided by tabs. The eSpace Tree is shown in Figure 1.4.

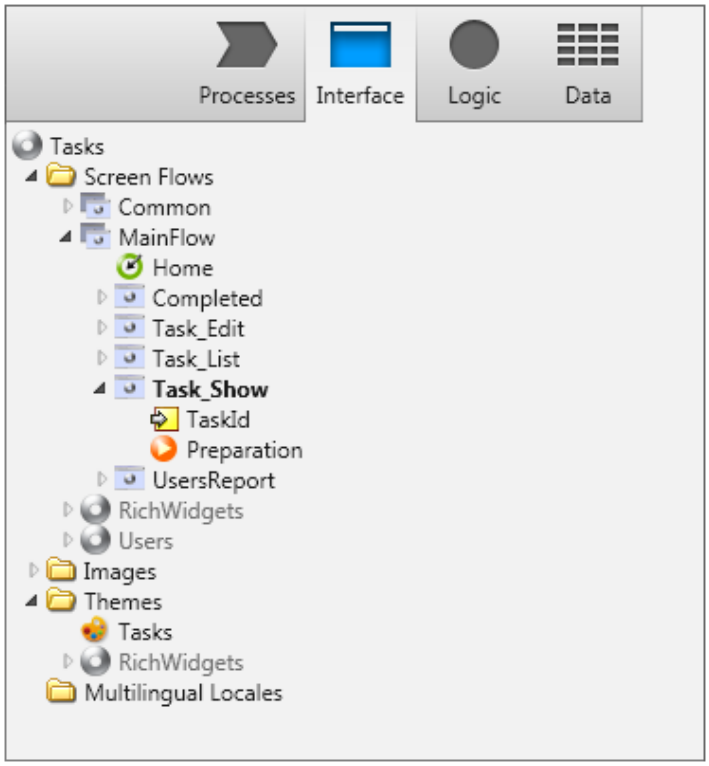


Figure 1.4: Service Studio: eSpace Tree.

The tabs divide the eSpace's components by categories, which are:

Processes components such as timers, scheduled jobs, among others;
Interface web-screens, images, pre-defined themes, etc.;
Logic functions, procedures, web-services and their web-methods, users and their roles, exceptions and more;
Data database tables, structures, session variables, entity diagrams, and others.

Objects have a tree structure, in which every object is the descendant of another object. Furthermore, the root of the tree is the node representation of the eSpace itself. Thus, the name "eSpace tree".

Regarding the tree representation, we proceed to explain with an example. Consider a web-page shown before, in Figure 1.2, "Task Show". This web-page shows the information of a task. It receives an input variable, *Taskid*, which uniquely identifies the Task to be shown. Thus, the input variable is a direct descendant (child) of the web-screen and the web-screen is a descendant of the eSpace node, which is the root node.

Properties Editor

Finally, we describe the Properties Editor. Every object in an eSpace has a set of properties, such as name, type (e.g., variables), value (e.g., expressions), among others. Once the developer has selected an object, the Properties Editor presents the properties of that object and enables the developer to change its values. Figure 1.5 shows the Properties Editor displaying the properties of the expression selected on Figure 1.2 (Title).

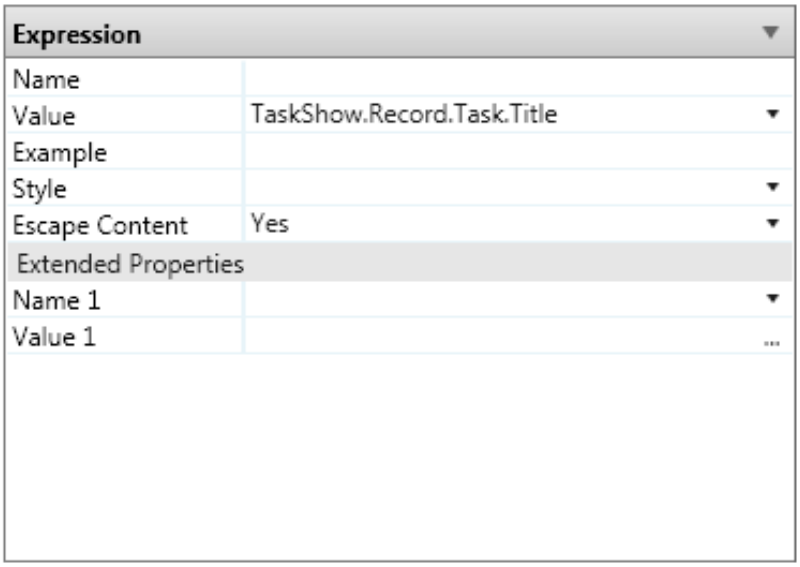


Figure 1.5: Service Studio: Properties Editor.

1.3.3 Platform Server

One of the components of the Agile Platform is the Platform Server. This component is responsible for the storage of the modules of an application, the compilation and deployment of the application and database. The Platform Server is the server to which developers commit their changes and it is through the Platform Server the developers test the application. From now on, we will refer to the Platform Server as the *development server*.

1.3.4 Existing Agile Platform Collaboration Model

The OutSystems Platform is ready for multiple developers working in the same web application. An application is composed by several eSpaces. From now on, we will focus our attention in the collaborative edition of a single eSpace. An

eSpace in commonly stored on the server side and developers have a local copy, which they change in isolation and then commit to the server.

Commonly, the development of a task is divided in 4 steps:

Contextualization The developer opens the eSpace and gains context of what is already done, in order to better understand what is left to be done. To do so, the developer might have to run the application for testing and/or debugging.

Development During this step, the developer performs the changes to the eSpace required by the task. This is done on the developer's version of the eSpace without changing the running version on the development server. The development step only ends when the eSpace is valid, i.e. the eSpace does not have compilation errors.

Testing Finally, after the development step, the developer usually needs to run a few tests in order to check if the behaviour and appearance of the application is the desired one. In order to run the applications with the local changes, developers must publish their version on the development server. This is due to the fact that the OutSystems Platform has a single execution environment, which is the development server. Uploading the version, compiling and deploying it are tasks automatically performed by the OutSystems Platform. The only required action from the developer is to actually command the Service Studio to initiate the publish process. After that, the developer runs the application on a browser. If the results were the expected ones, the developer can proceed to the closure of the task. Otherwise, another iteration of development-testing is performed.

Closure Closing the task does not require any more actions from the developer on the OutSystems Platform, as the changes were already saved on the development server on the test step. However, remote developers are not aware of the existence of these changes, unless they check for a new version on the development server.

Centralized Development Server

Each eSpace, on the OutSystems Platform, has a single development server. This development server is the only execution and testing environment, which is shared between all developers working on that eSpace.

Having a centralized development and testing unit has some advantages. For example, having one single central database shared across all developers

frees them from creating, configuring, loading and maintaining several individual databases. Loading a database with several gigabytes of information can be costly, especially if done several times. The same can be said regarding integration of external resources, which need permissions and configurations. It is simpler if they are done only once in a central system.

On the other hand, in order to test even the smallest of changes, the developers are forced to merge their work progress with other developers' work progress. This poses the risk of conflicts and, in case they happen, it breaks the focus of the developer. However, by forcing the developers to merge frequently, the time gap between one developer's changes and their effects on another's is reduced, thus reducing the correction effort cost.

Publish

Applications developed are executed in a central development server. Therefore, whenever developers want to test their an application, they are forced to publish it (commit changes) in the server. This poses a risk. When development is made in a collaborative way, there is the possibility that, when a developer wishes to publish his/her changes, another developer has already published some changes before and, in order to avoid losing work progress, it is necessary to merge the changes of both developers, before publishing. Merging the work of both developers is entirely of the responsibility of the last developer to publish his/her changes.

When the developer wants to commit, the Service Studio checks if anyone has published a version, i.e. if the current version on the development server is the same it was when Service Studio last updated its local version. If it is not, it means other commits were made concurrently, i.e. another developer published a new version, and changes have to be merged before publishing, in order to avoid losing work. After the merging process is concluded, the local version is published and overwrites the previously published version, becoming the new version on the development server.

Merge Process

When a set of developers concurrently edit an eSpace, for every commit that is done, a merge is required, except for the first commit and consecutive commits from the same developer.

When the developer commands the Service Studio to publish and it detects a

modified version on the development server, the Service Studio warns the developer and waits for instructions. The warning dialog is shown in Figure 1.6.

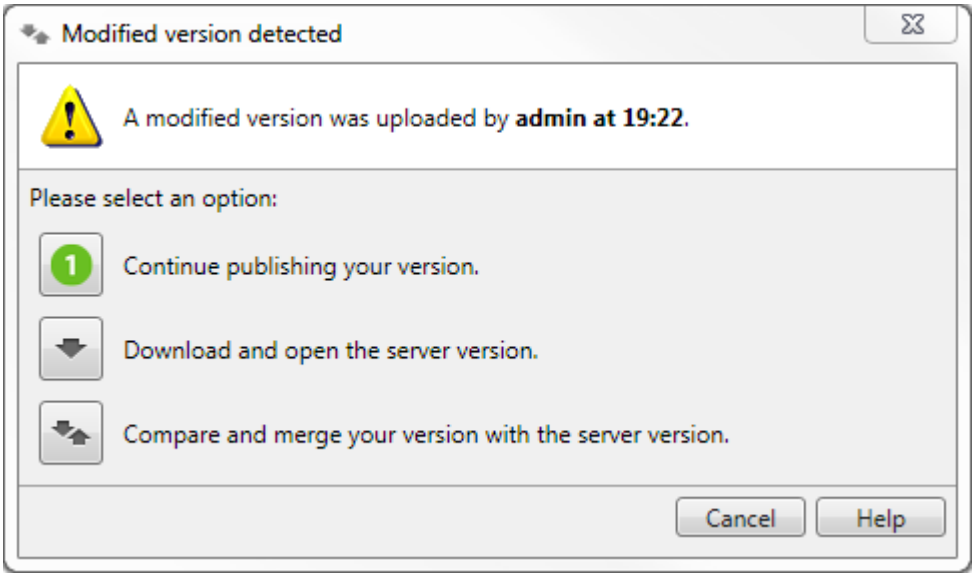


Figure 1.6: Service Studio: Modified Version detection warning.

The first option results in overwriting the version in the development server and, consequently, losing the work progress of other developers. The second option erases the local version of the developer and overwrites it with the development server version, which means losing work progress also.

Finally, the third option is the *Diff Screen*, which enables the developer to compare and merge both versions, as shown in Figure 1.7. The Diff method compares both eSpace versions and detects components that differ. Thus, on the Diff Screen, the developer is only presented with components that either are in a different state or do not exist on one of the versions.

The local version is displayed on the left side and the server version on the right. Service Studio informs that there are differences on a given component by setting the background color of that component to red. As we can see, the version of the "Homepage" Web-Screen is, in the local version of the module, different from the published one. The red background indicates that only one of the components can stay in the development server. In order to maintain the work progress from both developers, the developer currently publishing has to manually merge both changes in his local version and then publish again. This requires the developer to gain context of the work progress of another developer, understand his/her intentions, in order to merge without losing the work progress. This causes the developer to waste time he could spend on being productive and also causes a break on his focus and context. If the developer did not change the

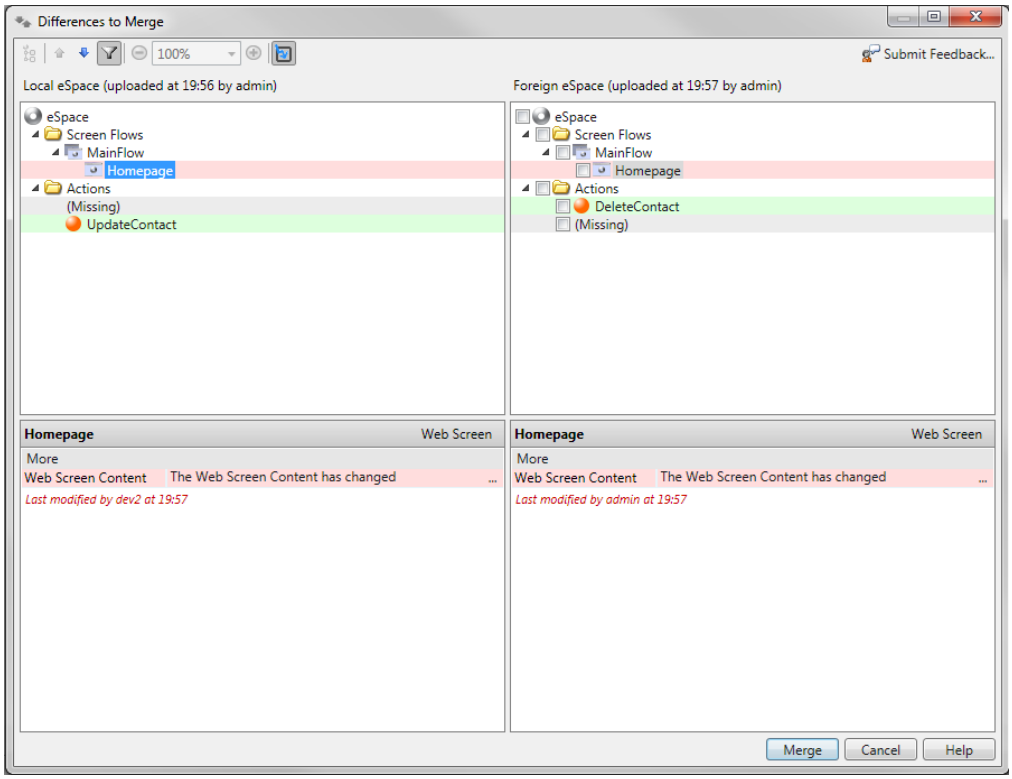


Figure 1.7: Service Studio: Diff Screen.

Homepage, then there is a difference because another developer has made changes and the local developer can simply opt for the published version of the *Homepage* web-screen.

Objects whose background colour is set to green can be merged without any trouble for the developer. An object marked as green means that the object only exists in one of the two versions. For example, considering Figure 1.7, the developer created an action called "UpdateContact", while another developer, concurrently, created an action called "DeleteContact". These actions can both be included into the resulting eSpace. The same would happen if both actions were already created and each of the two developers, concurrently, had deleted one of the actions.

In any of the previous cases, the developer has to explicitly inform which actions to maintain and which to delete. This is due to the fact that the Merge Process only *diffs* the two versions (the one published and the one about to publish). If we consider more than two developers working on the same eSpace, this work (merge process) has to be done by every developer that tries to publish.

As shown above, the merge process can be costly, especially if conflicts occur in more than one object. The fact that there is only one execution environment

forces the merge process to be unavoidable to the developers. On the other hand, in comparison with systems that make changes immediately visible to all users, such as Google Docs, despite the higher probability of conflicts, developers are not refrained from testing the application because of other developer's syntactical errors. This is due to the fact that, in order to publish an eSpace, the eSpace must be valid, i.e. must be *compilable*.

1.4 Summary and Approach

Information systems are critical nowadays. The time to market for software products is increasingly aggressive, thus the product development process becomes a team effort.

In order to be successful, team elements must coordinate their work efforts. To coordinate their efforts, they need to be aware of each others' actions and responsibilities. Furthermore, given the increasing physical distance among team elements, team awareness is damaged.

Moreover, we must consider the impact one team member's work progress has on another's. Conflicts happen when two, or more, developers work simultaneously on the same section of a given artefact. It requires time and context to solve these conflicts. In addition, it is possible that the solution to these conflicts leads to unexpected/unintended consequences, leading to work loss.

Our approach aims at mitigating the conflicts and the time spent to solve them. In order to satisfy this objective, we intended to design and implement a real-time collaborative software development feature on top of the OutSystems Platform. Furthermore, we intend to increase team awareness, to compensate the physical distance among the several developers of a software development team.

1.4.1 Thesis Outline

In the next Chapter, we will proceed to analyse some of the related work we studied in order to design our solution. In Chapter 3, we will discuss and present the solution to the real-time collaborative software development that we designed and implemented. Next, in Chapter 4, we will present our solution to increase team awareness, through visual details. In both those chapters, we will also discuss some chapter specific future work that we intend to address. Finally, in Chapter 5, we will address the conclusions and some relevant future work.

2

Related Work

Many systems have been designed to support collaborative editing of shared documents. Such documents can vary in type. For instance, raw text documents, programming language code, such as Java files, visual documents, such as an UML Diagram, among others.

Depending on the type of documents to be shared, there are different issues that must be dealt with. In this chapter, we will start by talking about some systems designed for collaborative editing and then we will show some of the techniques that were employed by these and other systems to solve important issues that arise in the context of collaborative work.

2.1 Groupware Systems

Groupware system can be divided in two main categories, depending on the way the work is shared: through asynchronous collaboration, on which users submit/check changes to/from a shared repository [Ced93]; and through synchronous collaboration, by allowing the users to see remote changes as they occur, such as in Google Docs. Then there are hybrid systems, which possess characteristics of both and we will designate them as Partially Synchronous.

The former case, upon synchronization, might lead to a larger amount of conflicts or duplicate work, i.e. two users producing exactly the same work, or one

user's work being contained in another's. For example, one programmer develops a method for an application and, concurrently, another programmer from the same team develops the same method, for the same application. Upon synchronization time, one of the developed methods is unnecessary and will be discarded.

On the other hand, fully synchronized collaboration might not be desirable in some contexts either. For example, on collaborative software development, one user's work might be at a state at which the project cannot be compiled and, therefore, another user's work cannot be tested.

Making only partial changes visible to others might be a solution for some cases but, for others, this may lead to intermediate states that are inconsistent in some manner.

2.1.1 Asynchronous Collaboration Systems

A common model for asynchronous collaboration for data access is the copy-modify-merge paradigm, in which a user gets its own private copy of the document, modifies it in isolation and later uploads his changes to be merged with the modifications concurrently produced by other users. Version control systems are a practical example of this model.

A version control system, in the practical role of Computer Science and Software Engineering, is a software that allows the management of different versions on the development on any document. These systems are commonly used in software development for controlling different versions - history and development - of the source code and documentation.

This type of systems is very present in companies and institutions of technology and software development. It is also very common in open source software development. It is useful in several aspects, both for personal use as well as small and simple to large commercial projects.

When using version control systems, users have a local copy of the document on which they work. The users perform changes on their local copies and are free to commit these changes to the remote repository or update their local copy to the one that is stored in the repository. Some version controls systems are CVS [Ced93], Apache Subversion [Sub], among many others.

2.1.2 Synchronous Collaboration Systems

Synchronous collaboration between a group of users consists of letting the users see the changes performed by other users of that group as they occur.

Google Docs is a SaaS that enables users to share and edit different types of documents, such as text documents, spreadsheets, forms, presentations, among others, providing synchronous collaboration. Users can access Google Docs via a browser, which frees them from installing specific software.

Google Docs, among other SaaS, is a proof that software is moving from the desktop to the Web. This is due to the fact that the Web allows ubiquitous and heterogeneous access, integration with other online services and avoids the installation and configuration of the software on the user desktop. These advantages also apply when it comes to software development tools.

Furthermore, it is important to take into account that the Web itself was conceived as a tool for collaboration, and most of the services and techniques developed for the Web are meant to facilitate collaboration. Given that, it is easy to conclude that moving other tools, like IDEs, from the desktop to the Web can increase significantly the collaboration between developers.

The Cloud9 IDE [IDE] is an example of such transition. Cloud9 IDE is a solution for collaborative software development. Developers work on a remote copy of the documents (stored on the server side) and the changes are propagated to all other developers working in the same workspace. Cloud9 also allows cooperation between developers, e.g. when a developer cannot solve/spot an error and another developers "intrudes" and helps solving the problem.

There are other relevant collaborative editing systems, although focused on collaborative editing of other types of documents. GRACE, the proof-of-concept presented in [SC02], is a system for collaborative editing of simple graphic documents, that tries to maintain consistency despite the presence of concurrent user operations that may conflict. This system is a solution for consistency maintenance that preserves all operations from the users. When two operations conflict with each other, the original object is duplicated and each of the conflicting operations is applied to a different version of that object. This multi-versioning approach to conflict resolution preserves all users' intentions and exposes the conflicts to the users, instead of cancelling some or all operations. Details about this solution will be presented later in this chapter, on section 2.2 (Technical Issues).

Collaborative graphical editing systems are often poor in terms of supported features, in comparison with single-user desktop applications. In [IN04], the authors propose a collaborative graphical editing system that provides *grouping* and *ungrouping* operations, besides the simple and common operations like *setColour*, *translate*, *scale*, *rotate*, etc. Grouping objects allows the users to perform actions on a group of objects, instead of performing the same operation several times in multiple objects. Although grouping objects might be very useful, it raises new conflicts to be resolved, e.g. a user (U_1) performs an action to change an attribute of a group of objects (G) while, concurrently, another user performs an action on a single object, belonging to G, in order to change the same attribute U_1 was trying to change. Details on how these problems are detected and solved will be provided further in this chapter, on section 2.2 (Technical Issues).

2.1.3 Partially Synchronous Systems

Collaborative editing applications are commonly classified as synchronous (in case updates become immediately visible to all) or asynchronous (in case users work mostly in isolation and propagate changes at specific points in time). Total isolation might lead to a larger number of conflicts. However, users might not want their work progress to always be visible to others, e.g. their work progress might not be relevant. In those cases, it might be desirable to only make partial changes visible to others. Next, we exemplify a series of systems that fall in this latter category.

In [Cam02], the authors present a solution for collaborative visual software development. The solution is extensible and applicable for different sets of visual objects with different syntax rules between them. CoDiagram, a proof of concept implemented by the authors, is a tool for designing Entity-Relationship diagrams. Therefore, the set of visual objects are the well known Entities, Relationships, Attributes, etc. For syntax purpose, when a system is implemented, rules must be defined, e.g. "an object of type `Attribute` must be connected to one, and only one, object of type `Entity`". There are two reasons why these syntax rules are necessary. One is to inform the user about syntactical errors. The other reason is for transaction definition.

A transaction, in their context, is a set of operations performed on an object, or set of objects, allowing the state of the program to become syntactically invalid until it becomes valid again. For example, creating a new `Attribute` will leave the program in a syntactically invalid state, since it's not connected to any

`Entity`. This results in the creation of a new transaction. When the developer connects the `Attribute` to some `Entity`, the program will reach a syntactically valid state again. This causes the end of the transaction and its propagation to the other developers collaboratively working on the same project.

Another partially synchronous collaborative editing system is VFC-IDE [MFV], but for Java projects. The system is a plug-in prototype for eclipse designed to allow collaborative editing of Java projects, aiming at reducing bandwidth usage, by postponing updates irrelevant for other users. This system is based on the VFC (Vector-Field Consistency) Model [SVF07], previously applied in multi-player games. VFC changes the degree of consistency, according to the locality-awareness techniques applied to each user. For example, in a multi-player game, a player vision about its surrounding should be consistent. On the other hand, the player does not need to have a consistent vision about distant areas, to which it would take several seconds, or even minutes, to reach.

VFC-IDE adapts this approach to software development. The locality notion is not about spatial position. In the software development context, position is the section of the code where the developer is working on. Distance is measured based on the relationship between constructs, such as class and interface hierarchies, methods, among others. This approach reduces bandwidth usage by not sending operations that need not be sent. Still, convergence is a goal, so changes shall be propagated, eventually. VFC-IDE defines a set of rules that trigger events to propagate changes to other developers, independently of the distance between the changes and the position of the remote developers.

Another important detail that should be mentioned is the fact that one replica does not propagate its user changes as they occur, even if the distance to other developers' position is short. Sometimes, changes might leave the code in a state that prevents it from being compiled. To solve this issue, the authors developed a *compilable state* detection mechanism, which is able to detect when the code produced is compilable or not. Upon a change, when the code is compilable, it is propagated (according to distance rules), along with the changes that were put on hold since the last compilable state.

This partially synchronous approach might be useful, but has its disadvantages. Not propagating operations that leave the program in a syntactically invalid state allows other developers to test their work progress. However, it may not be desirable as the developer might not be able to solve a problem that may be causing an error. In those cases, it would be useful and productive to provide

some degree of awareness to other developers and to allow another developer to see the incorrect state of the program and help solve it.

2.2 Technical Issues

In this section, we will discuss some technical issues that must be dealt with when developing a collaborative editing system. We will also present some systems and how these systems dealt with each issue.

2.2.1 Data Replication

On collaborative editing systems, each participant maintains a copy of the shared data and all updates are propagated to all participants. As in any replicated system, it is necessary that the replicas of the shared data are consistent with each other. The main difficulty to ensure consistency is to deal with concurrent operations. Executing operations, at all replicas, in the same total order guarantees that all states convergence to the same consistent state. However, on collaborative editing systems, this would require to delay the local operations from being executed until a consensus about the order was reached and the operations could be executed. Delaying local operations performed by the users would damage availability, thus damaging productivity, which are undesired effects.

A suitable approach is to execute local operations immediately and then propagate to all other replicas. However, on distributed systems without a central entity, executing remote operations when they arrive at a given replica can lead to divergent states. This approach is designated by "Last Writer Wins" policy, because executing operations in the order they arrive, in case of conflict, the last operation overwrites the previous ones. However, this approach can lead to undesirable behaviours, because the "last" operation at a given site might not be the same as at another.

Due to the varying communication latency, operations may arrive and be executed out of their natural cause-effect order. As shown in Figure 2.1, O_3 is generated after the arrival of O_1 at site 1. The effect of O_1 on the shared document has been seen by the user at site 1, when O_3 was generated (O_1 precedes O_3). Therefore, O_3 may be dependent on O_1, e.g. O_3 changes an attribute of an object created by O_1. However, O_3 arrives at site 2 before O_1. If O_3 is executed before O_1 at site 2, confusion may occur and may result in a final state of the shared data

different from other replicas. To prevent this from happening, a causal relationship between operations should be captured. Executing operations in a causal order assures that if an operation, O_x, depends on another, O_y, O_y will always be executed before O_x, at all sites.

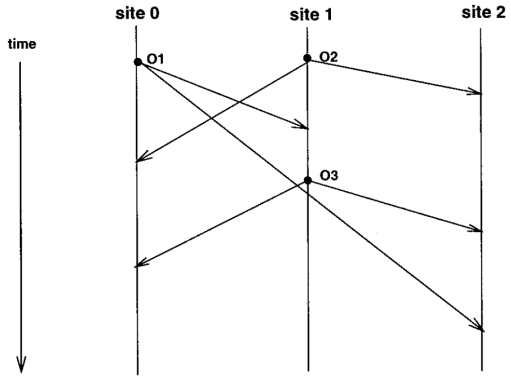


Figure 2.1: A scenario of a real-time collaborative editing session.

Capture Causality Information

To capture the causal relationship between all operations in a system, a time-stamping scheme based on a data structure - Version Vector (VV) [PPR+83, SM94] - can be used. Let N be the number of cooperating sites in the system. Assume that sites are identified by integers from 0 to N-1. Each site maintains an VV with N components. Initially, VV[i] = 0, for all $i \in$ {0 to N-1}, at all sites. After executing a local operation at site i, the value of VV at index i is incremented, i.e. VV[i] := VV[i] + 1. Afterwards, the operation is propagated to all other replicas in the system.

When a remote operation O from site s arrives at a given replica d, the operation's VV is compared with the local VV. The remote operation is said to be causally ready when the following two conditions are met:

1. $VV_O[s] = VV_d[s] + 1$
2. $VV_O[i] \leqslant VV_d[i]$, for all $i \in$ {0 to N-1} and $i \neq s$

The first condition ensures that O must be the next operation in sequence from site s, so no operations originated at site s have been missed by the replica d. The second condition ensures that all operations originated at other sites and executed at site s before the generation of O must have been executed at site d. Together, the two conditions ensure that all operations which causally precede O

have executed at site d. When an operation is not causally ready, it can be saved in history and postponed until it's causally ready. When an operation is finally causally ready, it can be executed on the local replica.

By comparing the VVs associated with two different operations, it is possible to capture a causal relationship between them. Given two operations, O_x and O_y, O_x precedes O_y, if $VV_x[i] <= VV_y[i]$, for all $i \in \{0 \text{ to } N\text{-}1\}$. If neither O_x precedes O_y, nor O_y precedes O_x, O_x and O_y are said to be concurrent, i.e. neither the effect of O_x was seen at site where O_y was generated, at the time O_y was generated, and vice versa.

This approach is useful and common on distributed systems without a central entity. VFC-IDE uses a similar approach to control concurrency over resources, such as files or folders. In U-Set, a data structure presented in [SPBZ11], version-vectors are also used for the same purpose. In [IN04] and GRACE, the same approach is used to capture the causal relationship between operations.

In systems that rely on a central server through which all communication is made, the causal relationship can be captured in a simpler way. For example, in revision control systems, an integer is assigned to each file/folder to describe its revision. When a user copies a file, along with the file, comes the revision of the file. After modifying it, when the user uploads it again, along with the uploaded file, the revision at which the file was when the user copied it, is passed and the file's revision is updated. Two updates have a causal order if the checked revision of one is greater-or-equal to the revision on which the other left the document. Otherwise, they're concurrent.

Concurrency Detection and Data Convergence Maintenance

Using the previously described approach of VVs, local operations can be executed immediately after their generation, thus availability and performance is improved; even though some remote operations may be delayed until all causally preceding operations have been executed. However, while this approach only preserves causality, it does not address the problems of divergence raised by concurrent operations. In those situations, conflicts may occur, i.e. the execution of concurrent operations in different orders might lead to divergent states.

Different systems have different possible conflict situations. For example, assume a shared text document initially containing the sequence of characters "ABCDE". Now, following the scenario shown in figure 2.1, suppose that O_1 intends to insert "12" between "A" and the rest of the sequence (O_1 = *insert*["12", 1]); and O_2 intends to remove the characters "CD", i.e. to remove two characters,

starting at position 2 (O_2 = *delete*[2, 2]). Although the combined effect should be "A12BE", if O_2 is executed after O_1, in site 0, the final effect will be "A1CDE", diverging from site 1.

Another example, assuming a graphical editing system: suppose O_1 intends to change the colour of an object *G* to red (O_1 = *setColour*[*G*, red]) and O_2 intends to change the colour of the same object *G* to blue (O_2 = *setColour*[*G*, blue]); O_1 and O_2 are causally ready for execution when they arrive at sites 1 and 0, respectively, but executing them on arrival will lead to divergent states, i.e. *G* will be red in site 1 and blue in site 0, after the execution of both operations in both sites.

To deal with concurrent operations, a simple approach is to impose a common order of the concurrent operations at all sites. This approach ensures convergence as all replicas witness the same effect of the operations. One possible way to do so is to save all operations, both local and remote, in a HB (History Buffer). That way, when a remote and causally ready operation arrives, it can be compared with other operations saved in the HB to check if it is concurrent with any of them and, in case it is, test if it can lead to a conflict. If it is not causally ready, it can simply be put on hold, but if it is causally ready, it can be concurrent to some already executed operations. For example, again, following the scenario shown in Figure 2.1, at site 0, when O_1 is generated and executed, the local VV is updated to [1,0,0] and O_1 is saved in the HB. Afterwards, when O_2 arrives with a [0,1,0] VV, following the rules shown above, O_2 is positively checked to be causally ready. Comparing O_2 with the operations in the HB, the system finds an operation, O_1, with which O_2 is concurrent and conflicting. The system then reverts the effect of O_1 (and, possibly, other operations executed after O_1, in the general case) and then decide on an order to execute the operations. This approach is somewhat similar to one approach mentioned before, Last Writer Wins.

There are other different techniques to solve conflicts in order to achieve convergence. Techniques such as Operational Transformation [EG89, SJZ+98], Multi-Versioning [SC02], among others. In the operational transformation approach, received operations are transformed according to local concurrent operations and then executed. Considering the previously given example of the shared text document, when O_2 arrives at site 0, it's compared against local concurrent operations, i.e. O_1. Since O_1 added 2 characters before the offset of O_2, the offset of O_2 must be incremented twice, i.e. O_2 := *delete*[4, 2]. However, building correct transformation functions is difficult [IMO+03].

The solution presented in [IN04] adopts an approach similar to operation serialization. As mentioned previously, the authors present a solution that allows the creation of groups of objects, allowing users to apply a single operation to multiple objects.

There are two kinds of conflicts in this system, real conflicting and resolvable conflicting: two real conflicting operations are those conflicting operations for which a combined effect of their intentions cannot be established, e.g. *setColour*(Obj_1, red) and *setColour*(Obj_1, blue); resolvable conflicting operations are those conflicting operations for which a partial combined effect of their intentions can be obtained by serialising those operations, e.g. O_1 = *setColour*(G_1, red) and O_2 = *setColour*(Obj_1, blue), where Obj_1 is contained in the group G. In this case, executing O_1 first and O_2 afterwards, a combined partial effect of both operations is achieved. This requires to undo O_2, apply O_1 and then redo O_2, at site where O_2 was generated.

When two operations are real conflicting, there are two ways of resolving them: cancel both operations (*null-effect* approach); or, based on previously defined global rules/priorities, decide which one of them effects and which of them is cancelled (*single-operation-effect* approach). For the distributed algorithm to know what actions to take on a conflicting situation, relationships between operations are statically defined. The system knows, for each pair of operations, what is the serialization order to be applied, if applicable, or which operation(s) to cancel, otherwise. The same serialization method is used for more than two operations.

GRACE provides a different solution based on multi-versioning [SC02]. Suppose we have an object, called G, and two operations, O_1 and O_2, that intend to move G to different positions, X and Y, i.e. O_1 = *Move*(G, X), O_2 = *Move*(G, Y), and $X \neq Y$. Some approaches to solve this would consist on a null-effect, cancelling both operations at the moment the conflict is detected. Others would prioritize one of the operations, based on some rule, and overwrite the other. To solve this problem, GRACE duplicates the object G, creating two objects, G_1 and G_2, and applies each operation on a different version of the original object. This example is shown in Figure 2.2.

Upon this event, users perceive the existence of a conflict. However, their intentions are still fully preserved. This would not happen in other approaches, such as null-effect, single-operation-effect, or even by serializing the operations.

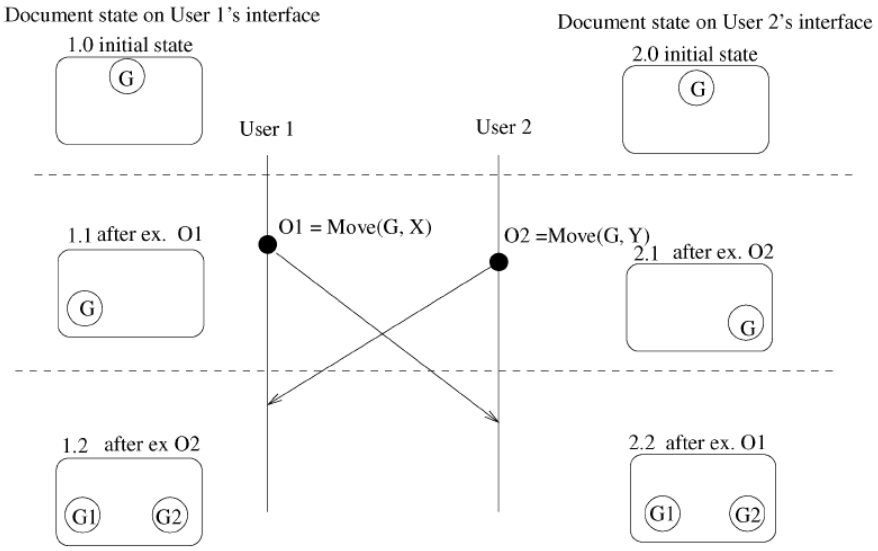


Figure 2.2: Conflict resolution by the multi-versioning approach.

Other Approaches

To prevent work loss, optimistic methods, such as those mentioned above, might not be appropriate for collaborative work, since there is a risk of having the system undo the user's work if later a conflict is detected. For this reason, the authors of CoDiagram selected a pessimistic locking approach [Cam02]. Whenever the user performs an action that triggers a transaction, a lock is acquired in the target object. The lock is released when the user performs an action that closes the transaction. It is also possible for someone to request a locked element from another user. This ability avoids deadlock situations where each user has locks on shapes needed by another user to complete the transaction.

The locking approach is pessimistic and focuses on avoiding conflicts between operations, rather than detecting and resolving them. However, it is possible that two users try to lock the same object concurrently. The authors are not very clear about how they solve this problem. Independently of how the system handles this situation and chooses to which user it shall give the lock, the users that were not able to get the lock are notified and able to continue their work.

There's another solution, which does not need to detect conflicts and, therefore, does not need to resolve them either: designing the system operations in a manner such that concurrent operations commute will avoid conflict detection

and resolution. This way, the only requirement is to preserve causality. One example of such approach is the TreeDoc [PMSL09]. TreeDoc is a CRDT[1] designed to manage a shared buffer, that relies on an binary tree. Concurrent operations on the TreeDoc are commutative. Therefore, given a combination of concurrent operations, the end result of the shared object is always the same, independently of the order by which the operations are executed, as long as causal relationships are preserved.

In VFC-IDE uses a TreeDoc to manage each Java file, thus operations are also commutative. However, the system protects the section of the code the developers are working on. The system creates *pivots* on the sections of the code where the developer is working on. VFC-IDE also allows developers to define explicitly their own *pivots* and remove them. The system assumes that a conflict occurred, every time a given developer performs changes in a line where another developer has a currently set *pivot*. Each time a conflict is detected, it is not automatically resolved by the system. The notification is sent to the developers' replicas and a dialog window (*Conflict View*) fades-in to alert the developers. With this careful approach, VFC-IDE does not assume anything about the conflicting operations and how the conflict can be solved, allowing the developers to solve their conflicts.

As shown in this section, conflicts may be dealt in different manners. We may adopt an optimistic approach and execute user actions locally when they're generated and propagate them to remote replicas. By doing so, we take the risk of having to *undo* some operations and, maybe, *redo* some of them to guarantee convergence among all replicas. On the other hand, this risk might not be desirable and one may want to rely on a more pessimistic approach, such as locking objects before editing them. The former approach is a better one in the sense that it provides better availability for the user. The latter avoids conflicts between operations, sacrificing availability. However, there's still the risk that conflicts happen on lock acquisition.

2.2.2 Awareness

Collaborative editing, as the name suggests, involves more than one user working on the same shared resource at the same time. In those systems, it is desirable that users have a sense of presence of other users, as well as who they are

[1] A Commutative Replicated Data Type (CRDT) is one where all concurrent operations commute. The replicas of a CRDT converge automatically, without complex concurrency control.

and what they are doing, i.e. what actions are they performing and what consequences/effects are they responsible for [GST05].

Awareness helps in avoiding and solving conflicts. Having this sense of presence, identity and authorship, users are more receptive to what they would call bugs. This sort of "anomalous behaviour" becomes the natural behaviour of the system on certain events. For example, suppose that there is a user that has no idea that there are other users working in the shared workspace. While he is working, he sees some of his work being undone, e.g. due to the occurrence of a conflict. Since the user is not aware of remote users and their actions, he might perceive this event as an internal error or bug. This penalizes the satisfaction of the user regarding the software. One user that is aware of the collaboration will be much more receptive to this kind of events. Awareness is also important when it comes to separation of roles. When a user perceives that another is changing a sector of the document, he can take that into account and work on a different section.

The systems previously presented in this chapter have some degree of awareness. VFC-IDE notifies the end developer when updates from remote developers arrive, so the developer can accept and let the updates take effect. Whenever a conflict happens during a work session, in VFC-IDE, the *Conflict View* fades-in and shows the conflicting areas to let the users revolve it. In GRACE, whenever a conflict happens and new versions of the conflicting operations target are created, the newly created objects are highlighted so the user perceives the occurrence of a conflict, as shown in Figure 2.3. In the system presented in [IN04], the users are always aware of the other users concurrently editing the same document. Furthermore, users are also informed by means of messages that appear on the lower part of the editor in the case that a conflict cancelled their operations. In CoDiagram, information about who had an element locked was provided, by tagging the elements or by colour-coding the elements.

Regarding awareness, Google Docs is probably, among all studied systems, the richest in terms of awareness. Google Docs allows the user to edit text documents, presentations, spreadsheets, among other documents. When editing any kind of text document, each online user is assigned a colour. The list of colours, and the corresponding usernames, is displayed on the top right corner of the window.

When editing a text document, inside the document itself, Google Docs provides information about users' cursors, coloured accordingly, as well as the selected text, if applicable. In case of the spreadsheets, Google Docs highlights the

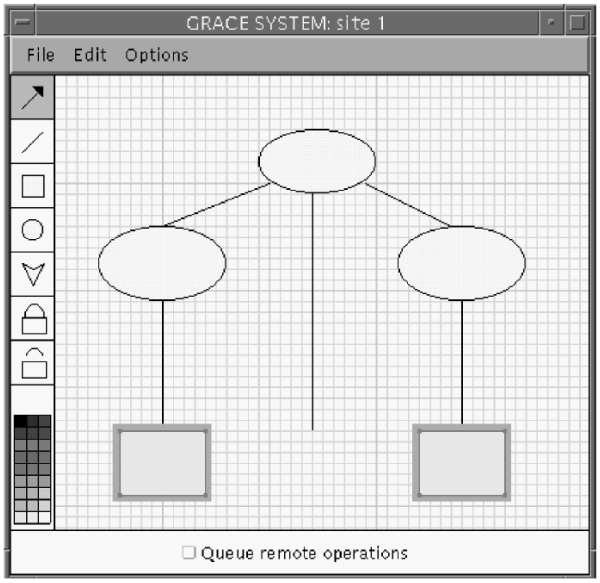


Figure 2.3: Providing awareness by highlighting conflicting objects.

cells selected by remote users by colouring their borders accordingly. When a user starts editing the value of a cell, Google Docs darkens that cell on the remote users' version. On graphic documents, like Drawing, Google Docs highlights objects selected by remote users by colouring their borders accordingly, just like in the spreadsheets. No information is provided about conflicts. When two users try to change the same property of the same object to different values, e.g. move an object to different places, Google Docs simply opts for one of the commands. This is no issue, since the users were previously aware of each other "being" in that object, thus being aware of the risks of editing that object.

2.3 Collaboration Models

In this section we will describe the collaboration model of two systems, Google Docs and Sky Drive. They are both collaborative editing systems of similar types of documents (text documents, spreadsheets, etc.). We will describe the user experience when working along with other users, possible operations and limitations.

2.3.1 Google Docs

Google Docs is a fully synchronous editing system, i.e. user changes performed to the shared document are *immediately* seen by other users concurrently editing the same document, keystroke by keystroke.

As in many SaaS, the access to the documents is made through the browser. Naturally, the documents are saved in the cloud. The typical use scenario is the following:

Opening User opens the Google Docs web-page and selects which document he wishes to edit.

Contextualization User gains context of what is done and other users are already doing.

Edition User edits the document. Conflicts are rare and almost the only way to make it happen is by performing actions purposely to generate a conflict. When those happen, Google Docs shows an error message reporting the problem, appealing to the user understanding and asking to try again.

Closure This step does not require saving the document for further edition. Saving is an action performed by Google Docs automatically.

Due to Google Docs being a synchronous collaborative editing system, the fact that the document is replicated across all users changing the document is almost unnoticed by the users, seeming a single document accessed by multiple users simultaneously. For the same reason, conflicts are rare. Under good network conditions, a keystroke is so quickly propagated, i.e. less than a couple of seconds, that in order to produce a conflict, two users would have to change the same section of the document in that short time frame. For example, consider a document with only the sequence of characters "ABCDE". Now, suppose that one user wants to replace the sequence "ABC" by some sequence of characters using the *paste* command and another user wants to replace the "CDE" by other sequence of characters also using the *paste* command. If they do it inside that time frame we mentioned before, it will raise a conflict and only one of the users will be able to affect the document.

2.3.2 Sky Drive

Sky Drive is a platform for collaborative editing provided my Microsoft. It is an asynchronous system, in the sense that merging other users' work progress is explicitly done by each user, although it provides some awareness regarding remote users' changes. The user experience is relevant in the context of this thesis, because one of our main focuses is to improve collaboration without affecting the user experience.

The common use scenario could be described as follows:

Opening User opens the Sky Drive web-page, selects the document he wishes to view and then clicks on "edit document".

Contextualization As in any collaborative work, before starting editing the document, the user needs to gain context of what is done what is left to be done. On the bottom right corner, the user can check who else is editing the document. If any of the remote users has performed changes to some section of the document, without saving them on the server, that section is locked, preventing the user from editing them.

Editing Users edit the document section they want, except for those which are locked by another user. When the user starts editing some section, the system locks that section on other users' replica. Like mentioned before, when the user changes some section of the document, a warning is propagated to all other replicas in order to lock that section and avoid conflicts from happening.

Save/Commit Being Sky Drive an asynchronous system, the users must explicitly commit their changes to the server by hitting the "Save" button, in order to save their changes. If another user had committed (saved) changes before, the work progresses of the local and the remote user are merged.

Merge Propagating the lock from a client to others takes several seconds, providing more time to produce a conflict. If two users edit the same section of the document, e.g. a paragraph, concurrently, when the second one saves, Sky Drive will inform the user that there is a conflict. Then, the user will only have two options: maintain the remote user's work progress or his own. In order to maintain both work progresses, the user will have to cancel the operation, copy his progress to some place else, for back up. Afterwards, the user saves the document and, in the merge operation, opts for the remote user's work progress to be maintained on that given paragraph where the conflict occurred.

Closure Closing the application without saving changes will result in losing them, like if it was on another software running on the desktop.

On Sky Drive, the "Save" action behaves similarly as the "Publish" action on the Service Studio, on the sense that its purpose is to commit changes to the server as well as check the changes performed by other users already committed. Although the purpose of the "Publish" action of the Service Studio is more than

that, one of the effects is committing the local changes and checking committed changes by other developers.

The main difference is the fact that Sky Drive provides information in almost real-time ("almost" because it still takes several seconds) about sections being edited by other users, and locks the sections in case the user did not edit that section yet. In case two users start editing the same section concurrently, neither of them is prevented from continue editing, i.e. the section is not locked in any of the replicas. This will lead to a conflict later on.

3

Collaboration Architecture

In this chapter, we explain our solution to enable Real-Time Collaborative Editing through the OutSystems Platform. In the first section, we give an overview of the architecture, regarding the main components and their tasks. Secondly, we describe some basic operations such as user entrance and exit and how the system behaves upon these events. Further, we will present our solutions to concurrency detection, conflict detection and conflict resolution. Then, we will present some performance optimizations and, finally, we will analyse and discuss some desired future work.

3.1 Overview

The system is composed by two major components: the Service Studio, which is the IDE through which the developers work, and the Service Center, which is the service through which the various IDEs communicate. The Service Center provides a Web Service interface to Service Studio. This Web Service provides several Web Methods, such as Upload, Download, Publish, among many others. Given that this Service was already defined, implemented, and tested, we decided to make use of it to enable more operations. These operations will enable real-time collaboration between the developers. The high-level architecture is shown in Figure 3.1.

In the current architecture of the OutSystems Platform, there is the notion of

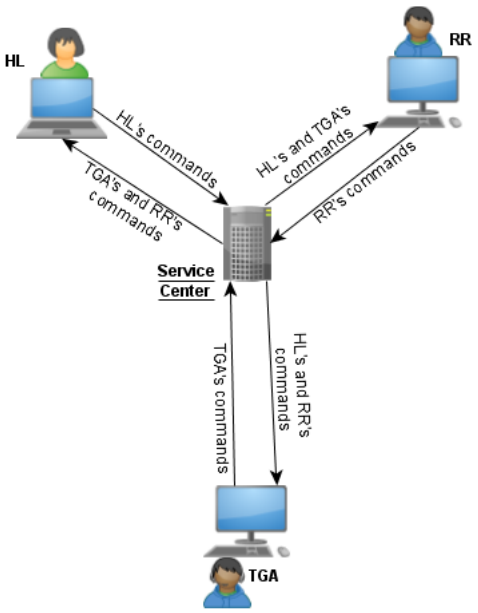
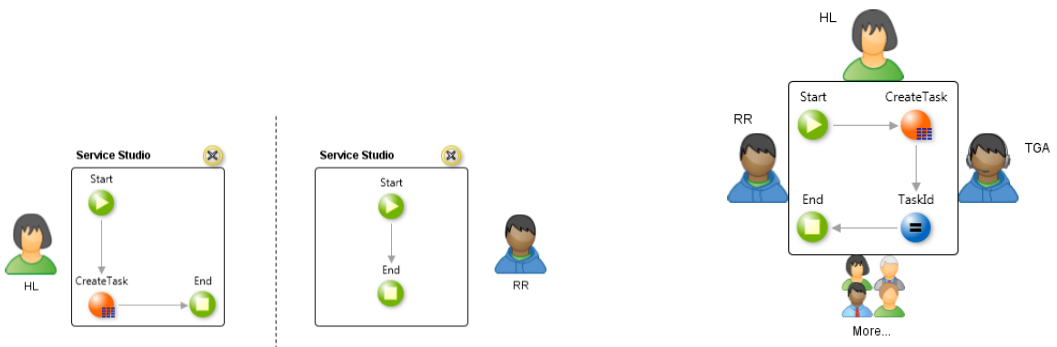


Figure 3.1: System Architecture.

a published version of the eSpace, which is the version that is compiled and running. The developers (more precisely Service Studio instance running on the developers side), before working on a given project, check-out its published version, edit it in isolation, and then, when desired, publish the new version (published version + changes). Therefore, developers have their own copy of the project stored on their local drives. Figure 3.2(a) illustrates this isolation. Our solution aims at giving all the developers the same version at all times (except for momentary inconsistencies), creating the concept of a global version, which we will designate as the **Working Copy**, as shown in Figure 3.2(b).



(a) Isolated editing of eSpaces

(b) Conceptual global model

Figure 3.2: Different types of collaboration

A simpler and more intuitive implementation would be based on a centralized architecture, although this architecture might raise some doubts regarding scalability. However, using the current centralized architecture to implement this concept does not pose as a problem, considering the development teams' size.

Furthermore, transposing communication responsibilities to the Service Center frees the IDEs from a considerable amount of computation, making them "thinner" and, therefore, more responsive to the end-user. In addition, and as it was said before, the OutSystems Platform relies on a centralized architecture. As such, to simplify the development path and for the remaining reasons explained above, we decided not to change that architecture choice.

3.2 Basic Operations and Communication Model

During the development cycle of an application, in order to ensure consistency among all replicas of an eSpace, there are several developer events that must be taken into account, namely events such as changes to the eSpace Model, user entrance and user exit. In this section, we will consider these events and describe the solutions we employed to implement them.

3.2.1 Commands

A single Platform Server, with one Service Center, is responsible for managing several applications and several eSpaces within the same application, as well as several developers.

To correctly capture the source of a given command, we must uniquely identify developers. For example, when one developer executes one command, Service Studio serializes the command and sends it to the Service Center. Afterwards, the Service Center needs to send that same command to every other process/machine for which the command is relevant, i.e., that is editing the same eSpace to which the command concerns.

In order to differentiate the several Service Studio instances, each instance must be uniquely identified. The OutSystems Platform allows two different developers, in two different machines, to login with the same credentials, i.e., username (and password). Therefore, the session username is not enough to uniquely identify the developers, or their Service Studio instance. Moreover, one developer can open several Service Studio instances, with different credentials, in a single computer. A single developer can even open several tabs inside the same Service

Studio, with all the tabs editing the same eSpace. One could struggle to understand why a developer would open two Service Studios, or two tabs, to work on the same eSpace. Nonetheless, for a matter of consistency, it is a possibility that must be dealt with. To deal with the problem, every Service Studio tab generates a unique key which we will designate as **Instance Identifier**, or **Instance_id**.

To store commands, the Service Center has a database table, designated **User_Commands**, which consists of the following attributes:

id an auto-number generated when a tuple is inserted;

eSpace the eSpace identifier to which the command concerns;

Instance_id the Service Studio tab identifier (where the command was generated);

UserName the username of the user who generated the command (for presentation issues);

Command the serialization of the actual command.

To enable communication between the several IDEs and Service Center, a new web method was implemented on the Service Center. This web method enables the IDEs to send the list of locally executed commands (even if empty) and to check for remotely executed commands, so that this information can be exchanged among IDEs periodically. The web method has the following signature:

Inputs

- eSpaceKey: String
- Instance_id: String
- UserName: String
- Commands: List<String>

Outputs

- RemoteCommands: List<Pair<Integer, String»
- CommandsOrder: List<Integer>

Thus, the IDEs frequently (e.g., every 500 ms) send locally executed commands and check recent commands that were executed remotely. In order to decide which commands to send to a given instance, the Service Center must

know, at any moment, which commands that instance has executed. The **id**-attribute is an auto-generated number which increases with the number of insertions. Therefore, for two commands, cmd_1 and cmd_2, if cmd_1 was inserted before cmd_2, cmd_2's id will never be greater than cmd_1's id. Taking this into account, it is trivial for the Service Center to infer which command was last seen by an instance. At the end of the web method, before returning, the Last Seen Command by the instance is the greatest id among all relevant commands' id. The relevant commands are the ones that the instance sent as input and the ones that it is about to receive as output. From there, the web method stores the data in a database table, designated **LastSeenCommands**(Instance_id, LastSeenCommand). Therefore, when the next polling occurs, the Service Center will only retrieve the commands whose id are greater than the Last Seen Command stored at the database table, for that instance. The entire sequence of steps is shown in detail in Figure 3.3.

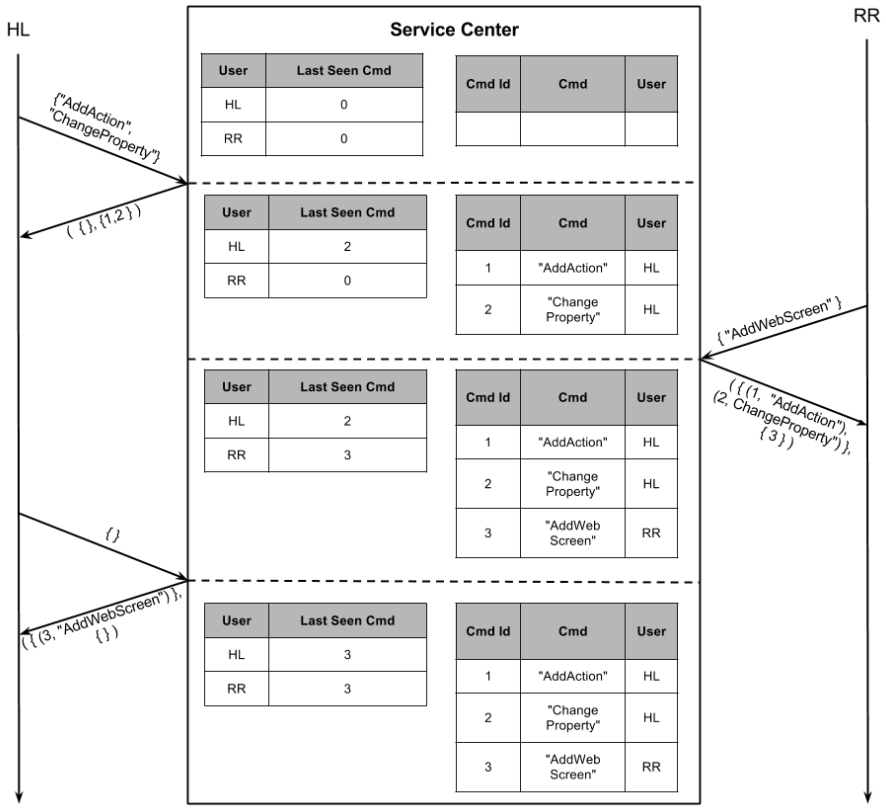


Figure 3.3: Example of a polling session.

The first output variable, RemoteCommands, is a list of pairs, since every

command has an order assigned (which corresponds to its id at the database table). The second output variable, CommandOrders, is the list of orders corresponding to the commands sent as argument for the fourth input variable, Commands. These commands are relevant for conflict detection and resolution, which will be discussed further in this chapter, in section 3.4.

This polling approach presents a communication overhead, since the web-method is called frequently, even if there are no commands to be either sent or received.

As a possible alternative solution to this problem, when a command occurs at a given Service Studio instance, that instance immediately sends the command to the Service Center and the Service Center then propagates it to every other Service Studio instance immediately. With this approach, the IDEs would not have to communicate with the Service Center to check whether or not there are remote commands to execute. However, we cannot guarantee that the Service Center machine can initiate a communication session (via socket) with all Service Studio machines, e.g., due to the corporate firewalls.

Yet another alternative solution would be to simulate a connection initiated by the Service Center. The IDE would have a dedicated thread that made a web-method call. The Service Center would not reply, leaving the IDE thread blocked. Whenever the Service Center had a command to send to that specific IDE, the Service Center would send the needed data through the opened channel. This would avoid the polling approach, where the IDEs fetch the remote commands, instead of receiving them when they occur. However, it is not entirely feasible, due to the fact that some network components, such as firewalls or routers, buffer the data before transmitting, until the data reaches a specific size or the server replies.

As mentioned above, there are other alternatives regarding the communication architecture. However, besides the reasons explained above, we opted for the polling approach due to reduced time regarding the implementation of the proof of concept. Moreover, there would be a potential need to make changes on the Platform Server and, in practice, the polling approach has revealed to be good enough.

3.2.2 User Entrance

When a developer opens an eSpace, that developer must be provided with the same view of the eSpace as all other developers editing that same eSpace, i.e., the working copy. This working copy is represented by a baseline (static eSpace

version) and a list of commands that have taken effect on that baseline. For the baseline, the published version can be used. Therefore, when a developer enters the developers' group editing that eSpace, the Service Center provides the IDE with the published version along with a list of commands. Upon receiving the baseline and the commands, the IDE executes the commands on the baseline and presents the result to the developer, in order to start editing.

From the Service Center point of view, just before returning the eSpace and the list of commands, an entry is created in the database table LastSeenCommands. That way, when the developer's IDE starts the polling thread, the Service Center will already know which commands it must send.

3.2.3 User Exit

The solution we present shifts the responsibility of initiating the communication to the IDEs. Therefore, immediately after a developer closes the Service Studio, the polling thread is automatically stopped. This does not require a special attention from the Service Center. However, it does require some attention from the IDE. For example, if a developer executes a command and immediately closes the IDE, there is a chance that the recently executed commands were not propagated to the server yet. Therefore, the IDE should check, before closing, if there are commands to be sent and, if there are, send them. Yet, not performing this check would not risk consistency: it would just represent a work loss, since neither Service Center, nor any online Service Studio instance, would be aware of these commands.

There are other issues regarding developer connectivity that are relevant to this collaboration model. During a development session, one developer may have connectivity issues or may even completely lose the network connection. During these periods, the developer would produce work progress that could conflict with other team element's work, like in an asynchronous solution. However, since software development in an intense collaboration model, such as ours, demands connectivity, we decided that offline development would not be considered in our use-case scenario.

3.3 Concurrency Detection

Developers are free to execute commands, independently of other developers' activity. Therefore, it is acceptable and expected that one developer executes commands without knowing that other developers are also executing commands. We

consider that two, or more, commands are concurrent when they were executed without the knowledge of each other's existence. More precisely: one command (e.g., cmd_2) follows another (e.g., cmd_1), when cmd_2 is generated after the arrival of cmd_1; therefore, when cmd_2 was generated, its corresponding developer had already seen the effects of cmd_1; given two commands, they are said to be concurrent, whenever none of them follows the other.

Mishandled concurrency can originate inconsistency among replicas of the eSpace, as merging concurrent commands to produce the final state of the project can be complicated when those commands are conflicting. In the next section, we will discuss when concurrent commands can lead to conflicts and how these can be detected and resolved.

To detect the occurrence of concurrent commands, Service Center resorts to the LastSeenCommands database table. In particular, when an instance polls the Service Center, it can infer that there were concurrently executed commands if: there are commands on the User_Commands database table whose id is greater than the value on the database table LastSeenCommands for that Service Studio instance; and that instance passed a non-empty list of executed commands as argument. Consider the example given on Figure 3.3. When the Service Studio of the developer named RR polls the Service Center to send the command "AddWebScreen", it is clear that some commands were executed concurrently: there are commands on the database table which RR's Service Studio had not executed yet (its Last Seen Command was 0), namely commands "AddAction" (whose id is 1) and "ChangeProperty" (whose id is 2); and RR's Service Studio sent a non-empty list of commands, i.e., "AddWebScreen". Therefore, command no. 3 is concurrent to both command no. 1 and no. 2.

However, concurrency does not always originate conflicts. Once more, consider the example given above. If one developer creates a Web-Screen and another developer, concurrently, changes the name of a User-Action, independently of the order by which these commands are executed at each replica, the replicas will converge and become consistent as soon as they execute both commands. In other words, some commands are commutative among each other. In those cases, we can merge the concurrent commands to produce the final state.

3.4 Conflict Detection and Resolution

It is possible that, for two (or more) commands, executing them in different orders produces different resulting states. When that happens, we designate it by

"conflict". Mishandling conflicts might lead to divergent states among the eSpace replicas. Let us illustrate with an example. On Service Studio, when a developer creates a User-Action, the User-Action has a default name ("Action1" if there is no "Action1" yet; "Action2" if there already an "Action1" and no "Action2" and so on). If one developer creates an User-Action and, concurrently, another developer creates another User-Action, both will be named "Action1" in their respective instance. After the propagation of commands, if the remote commands are executed blindly, each instance will have two User-Actions ("Action1" and "Action2"). However, the "Action1" of one instance will correspond to the "Action2" of the other replica, and vice-versa.

Executing the commands in the same global order would ensure that the instances' eSpace would match with each other. Moreover, on the User_Commands database table (stored on the Service Center), the commands' id attribute represents the orders by which the commands arrived at the Service Center. Thus, the global order by which conflicting commands should be executed is defined by the Service Center. In other words, for two commands, cmd_1 and cmd_2, if cmd_1's id is greater than cmd_2's id, in case of a conflict between cmd_1 and cmd_2, cmd_1 must be executed after cmd_2 at all Service Studio instances. This rule ensures that all instances convergence to the same state.

To implement this conflict detection and resolution strategy, each Service Studio instance polls the Service Center, every 500 ms. At each polling step, the instance sends the commands locally executed in the last 500 ms and receives the remotely executed commands that arrived at the Service Center during the last 500 ms. Along with remote commands, the instance receives the order by which the commands, local and remote, should be executed. Then, to ensure convergence among replicas, there are two possible approaches:

Optimistic Assume conflicts are rare. Thus, after each polling step, compare the local commands with the remote commands and check if they are conflicting. If they are, undo the the locally executed commands and, afterwards, (re-)execute the commands respecting the order assigned by the Service Center.

Pessimistic Assume conflicts are frequent. Thus, after each polling step, instead of wasting time and computation checking for conflicts, since they are frequent, assume they happened and proceed with undoing the locally executed commands and executing all commands respecting the order assigned by the Service Center.

The optimistic approach would avoid undoing commands, when unnecessary. However, it would require the implementation of complex rules to define which commands raise conflicts and in which circumstances. It would require to match every single command with every other and define if those two commands are always conflicting, never conflicting or just in some circumstances, and which. For example, the "AddUserAction" command conflicts with itself, in case they are executed in different orders in two different replicas, the result would be two actions with switched names. On the other hand, creating a User-Action will never conflict with the creation of a Web-Screen. However, it would not be always so simple to decide whether two commands conflict with each other or not. For instance, consider a command at one replica, cmd_1, to create a User-Action and another command at another replica, cmd_2, to rename some existing User-Action. Let us also assume there is no User-Action named "Action1". Therefore, cmd_1 would result in the creation of an User-Action called "Action1". Furthermore, cmd_2 would only conflict with cmd_1, if the developer tried to rename some User-Action to "Action1", which is the default name for the action created by cmd_1. Besides the implementation of rules for all possible pairs of commands, the optimistic approach would require runtime verification.

The pessimistic approach is simpler and safe as it ensures that all commands are executed in the same order at all Service Studio instances and, therefore, there is no possibility that the replicas of the eSpace diverge. Furthermore, the implementation costs would be very low. Due to these low implementation costs, in our proof of concept, we opted to implement the pessimistic approach. The pessimistic approach showed insignificant runtime costs and has proven to perform well and to be responsive.

3.5 Optimizations

The solution that we presented enables the developers to freely open and close the eSpaces without losing any work and without needing to upload or publish. The **working copy** is stored at the server side, being the concrete representation of the abstract concept, which is the global vision of the eSpace. When the developers open the eSpace, they are provided with that working copy, which is the one they change. Furthermore, when they close their respective IDEs, the working copy is already stored at the server side. This working copy is represented by a static eSpace (baseline), which is not changed, and a list of commands. However, the list of commands grows over time. Eventually, the list becomes big enough to

the point of deteriorating the IDE responsiveness, upon developer entrance, since the IDE must execute all commands on the received eSpace, in order to obtain the same state as other IDE instances.

Clearly, the Service Center should frequently advance the baseline and delete no longer needed commands. More precisely, the Service Center should update the stored eSpace version and delete the commands that already took effect on that eSpace version. Since Service Center knows which developers have executed which commands (table LastSeenCommands), it should be trivial to decide which commands are to be deleted. In other words, all commands that have already been executed by all developers working on a given eSpace and have already been executed on the eSpace version stored on the server side, can be deleted. Thus, this raises the need to determine which developers are still online. For example, suppose that after polling the server, the last executed command by one developer, e.g. *jan.jokela*, was the command no. 56. Afterwards, *jan.jokela* closes the IDE. Minutes later (with more commands executed by the group), when the Service Center is about to advance the baseline and delete some commands, every instance has executed all commands until the 90th command, except *jan.jokela*'s instance. However, if the Service Center cannot detect that *jan.jokela*'s instance is offline, it cannot delete all commands until the 90th.

To detect when a given developer is online, we designed a database table, designated **HeartBeats**, composed by two attributes, **Instance_id** and **LastHeartBeat**. The purpose of this table is to keep track of when was the last time there was any sort of communication between the Service Center and the several Service Studio instances. Given that the Service Studio instances are supposed to communicate with the Service Center every 500 milliseconds, at any moment, the Service Center can infer that some Service Studio instance was closed: this is the case when its last heartbeat happened too many seconds before. During 30 seconds, a Service Studio instance polls the Service Center 60 times. If, during that time, there is a Service Studio instance that did not poll the Service Center once, the Service Center infers that the Service Studio instance is offline.

The Service Center does not have, at any time, the eSpace loaded in memory and, therefore, cannot execute commands. The solution we designed to advance the baseline, i.e., update the eSpace version stored on the server side, lies on the IDEs. During periods of inactivity, i.e., a few seconds without any local or remote commands, the IDEs send their version of the eSpace to the Service Center, in order to advance the baseline. The Service Center, upon receival of an eSpace version from a Service Studio instance, can infer which commands were executed

on that version, since the Service Center knows which commands were executed on the Service Studio instance that sent the updated eSpace version. Afterwards, the Service Center stores the updated version and deletes irrelevant commands. The commands that are irrelevant are those that have been executed on the updated version and that have been executed at all instances that are still online. Due to different network conditions, some instances might take more time to poll the server and, thus, be behind schedule. On those cases, the Service Center must keep the commands stored.

3.6 Key Consistency

The solution we presented so far is a general solution, as it ensures consistency among all replicas of a given model. However, in the context of Software Development through OutSystems Platform, there is one more requirement to be accomplished.

Every object in the OutSystems Platform Model has a Key, which uniquely identifies that object. These keys are generated automatically by the programming language supporting the implementation of the Platform. They are generated at the constructor of the abstract class, AbstractObject, which all object implementations extend. Therefore, even if all Service Studio instances execute the same set of commands in the same order, the keys for each object would still be different. If we guaranteed consistency among all replicas, without guaranteeing key consistency among all Service Studio replicas, all replicas would possess a copy of the eSpace, such that all copies would have the same visual appearance and the same functional behaviour. However, these eSpaces would not be equal among each other, since for each object on both eSpaces, even if the rest of the fields of the objects would match, if their Object Key did not, they would still be considered different objects.

Solution

To accomplish this requirement, we extend the command serialization, in order to transfer information regarding the keys of the objects created by the commands. Thus, when a command is on local execution, i.e., the command is being executed due to the developer's commanding, then the generated Keys are stored in the command serialization. On the other hand, when a command is being executed because it is a remote command that was propagated, since it will originate the

creation of the same objects, the necessary keys are already present in the command serialization that is received. Thus, the objects, upon creation, check their key on the information received from Service Center.

Key Structure

During the playback of a remote command, the created objects do not have a key until they check their key on the key structure present on the command serialization. The creation of the objects follows a deterministic order, i.e., the same command, executed at different instances, creates the same objects by the same order. Therefore, recording the keys in a happened-before order would suffice. However, for a matter of system robustness, we opted for a more complex approach.

The eSpace Model is represented in a tree, where every object has a non-null parent, except for the eSpace node itself, which is the root of the tree. Furthermore, every object has a type which can vary. It can be *UserAction*, *InputParameter*, among many other possibilities. Thus, in the key structure saved on the command serialization, the object is identified by its parent's object key and its type.

To clarify this point, we will illustrate it with an example. Consider the command *AddUserAction*, which creates an User Action and creates a Start and an End node in the User Action Flow. The *AddUserAction* command generates a script similar to the following:

```
<KeyStructure>
  <Entry ParentCompoundKey="/" ChildKind="Action">
    <Key Value="F0+Cw" />
  </Entry>
  <Entry ParentCompoundKey="/UserActions.F0+Cw"
         ChildKind="Start">
    <Key Value="j8zw" />
  </Entry>
  <Entry ParentCompoundKey="/UserActions.F0+Cw"
         ChildKind="End">
    <Key Value="2qkw" />
  </Entry>
  <Entry ParentCompoundKey="/UserAction.F0+Cw/Nodes.j8zw"
         ChildKInd="connector">
```

```
    <Key Value="Nx0g" />
  </Entry>
</KeyStructure>
```

During the recording of a command, the objects (at the constructor of Abstract Object) will generate the keys and store them in a dictionary, which is later serialized to a key structure similar to the one presented above. During the playback of a command, the serialization is parsed to a dictionary, from which the objects check their respective keys.

In some cases, one command might result in the creation of multiple objects of the same type, which are children of the same node. In those cases, we cannot differentiate those objects, since they both have the same type and parent, and there is no more variables we can consider. Thus, in order to enable the recording of the object's keys, instead of having just one key for each pair of (ParentCompoundKey, ChildKind), we have a list of keys, whose creation follows a deterministic order, which we can take advantage of. Given that the creation of those objects follows a deterministic order, we implemented a solution such that the recording of the objects' keys follows the same order. So, when playbacking the creation of the objects, given the fact the order is deterministic, the playbacking will follow the order presented on the list of recorded keys.

With this solution, we were able to ensure that all replicas of an eSpace have the same object identifiers for the same objects. Thus, more than having replicas with the same appearance or functional behaviour, we have replicas that are equal to each other.

3.7 Limitations and Future Work

The proof of concept is simply a prototype, which requires further work. Therefore, it has some limitations. In this section, we will discuss some of these limitations and suggest some relevant technical issues that should be addressed in the future.

Command Serialization

The command serialization used on our proof of concept was the one already implemented in the OutSystems Platform, as it was needed before for session recording and playback. The Event Recorder enables the developer to record a

session of operations performed by an eSpace, which generates a script for further replay (through the Event Playbacker). We used two components to serialize this commands, by starting a recording session as soon as the developer orders a command to be executed and stopping the recording session as soon as the command is complete. However, the Event Recorder and Playbacker were not implemented considering real-time collaboration between developers. In particular, the command serialization is not independent and requires the right presenter to be opened and to proceed with the command execution. This causes flickering, as the screen of one developer changes (even if for less than half of a second). For example, if a developer is working on a web-screen and his/her Service Studio receives a command to change something in the flow of an action, the Service Studio will open the action, proceed with the change and then re-open the web-screen. In the future, the command serialization should be independent, such that the commands can be executed strictly in the Model layer of the Service Studio.

Undo and Redo Operations

Going back through the state of a document using the *Undo* command of an editor is a common action taken by users. However, enabling this command in an environment where the collaboration is made in a real-time manner raises some issues. For example, consider that there are several developers editing the same eSpace, in different computers, without communicating with each other. Assume that two of these developers are named John and Jane. John creates a User Action, named "Action1" by default. Jane perceives and renames the action to "Johns Action". A couple of seconds after the propagation of the command from Jane's computer to John's computer, John hits *ctrl+z* (shortcut for Undo).

The issue is determining the right Service Studio behaviour upon this event. There are two possibilities: first, undo the command that was last executed on John's Service Studio, regardless of the developer who ordered its execution; or second, undo the command that was last executed by John (the creation of the user action). Undoing the last executed command, regardless of the developer who ordered its execution, may result in undoing a command that John did not intend do undo, and may result in undoing the command of another developer regarding a completely different section of the eSpace.

Undoing the command that was ordered by John requires undoing every command that depend somehow on John's command. In this case, it is simple: undo Jane's command and John's command. But consider a different scenario: John

changed the name of a variable from "colour" to "isRed". Afterwards, Jane changed the variable type from "String" to "Boolean", based on the new name of the variable. The issue here is what happens to Jane's command if we want to undo John's command. Jane's command was causally dependent on John's command. In other words, at the time Jane ordered the execution of her command, Jane's Service Studio had already executed John's command. As far as we know, Jane could have seen John's change and maybe her command was a result of what she saw. Maybe it was not. However, all we (as a Service Studio) can capture is the causal relationship between commands. We cannot capture the intentions of the developers. If her intention was based on John's command, we should undo both. If it was not, we should only undo John's command.

Nevertheless, if we want to provide the developers with the option to undo and redo operations, this is an issue that requires analysis and decisions to be made, regarding what effect should an undo have on posterior commands.

3.7.1 Error Containment

This new way of collaboration brings the developers closer to each other, as if they were editing the same shared eSpace. However, this proximity poses a relevant issue: syntactical errors. Since all commands and propagated, one command that creates syntactical errors will leave all developers with error messages. Firstly, this is distracting. Error messages are visually notable. Furthermore, developers would feel tempted to read the error message and try to understand where or why it happened. These insignificant seconds will be enough to distract the developer and damage his/her productivity, since he/she will need a few minutes to recover the full attention and productivity he had before the distraction [PR11]. Secondly, and most importantly, it is impossible to test an eSpace with syntactical errors. In other words, since we are propagating an error to all developers, we can create a situation where we stopped all developers from testing their changes, due to the actions of a single developer.

The approach we considered the most appropriate is based on command propagation containment. In others words, the Service Studio does not propagate a command to Service Center if the local eSpace has an error. Instead, the Service Studio will accumulate commands until one "corrects" the errors. When that happens, the Service Studio will propagate all contained commands in bulk. This approach prevents remote developers from being distracted due to errors caused by a third party and it enables remote developers to test their changes. However, this approach must go through further analysis as it can be naive. For example,

although an error might be triggered by one developer, it might have been previously and silently introduced by another.

3.7.2 Fault Tolerance and Recovery Protocols

During the development of this thesis and its proof of concept, we assumed that there would be no problems regarding the computers and network behaviour. Therefore, we did not analyse nor developed any kind of fault tolerance and recovery protocols.

In the future, we should consider such scenarios and develop adequate solutions. For an example of a such scenario, we can consider a communication failure between a Service Studio and Service Center, after the Service Center has sent a reply with remote commands. In more detail, the Service Studio calls the Service Center web method to download remote commands; the Service Center replies with two commands, with ids 34 and 35; due to network problems, the Service Studio does not receive the message; the next time Service Studio communicates with the Service Center, the Service Center will only send commands with ids greater than 35 and, therefore, commands 34 and 35 will not have effect on that particular Service Studio, leaving its eSpace in a different state than the corresponding eSpace on the other instances. This is problematic, since the Service Center is the component that knows which commands it sent to which instances. This could be resolved by keeping this information on the Service Studio side and add an extra argument to the web method. In other words, each Service Studio would know what was the id of the last command it executed; and the Service Studio would ask specifically which commands it wanted.

The previous example is just an illustrative scenario. There are other scenarios that should be considered and analysed.

4

User Presence Awareness

Awareness is defined as an understanding of the activities of others, providing a context on one's own activities [DB92]. In software development, "awareness" can be described as as a team member's knowledge of what other team members are working on and what how their activity can impact his own.

More than bring a more intense form of collaboration between the developers, we wanted to give all developers information regarding their colleagues' presence and actions. Giving the developers awareness of which objects, actions or web-screens their colleagues are editing, will encourage developers to work in a different section of the eSpace, in order to avoid conflicts. Even if the applications development would not be performed in a real-time collaboration manner, even if it would be performed in isolation, team awareness would allow the developers to work in distinct sections and, therefore, help them to come up with strategies to mitigate the amount of time used during the merge process [DISK07].

In this chapter, we start by presenting the different forms of awareness given to the developers and their contributions. Later in the chapter, we will illustrate the methods that were used to achieve the final result. Finally, we present an optimization to eliminate communication overhead.

4.1 Forms of Awareness

Previously, we explained that the Service Studio is divided into three main sections: the Content Editor, the eSpace Tree and the Properties Editor. In this section, we will present how we give awareness to the developers in some of these Service Studio's sections.

4.1.1 Global Awareness

Global awareness allows the developers to perceive the presence of their colleagues, regardless of their context. The global awareness given, as the name suggests, is the most general and simplest awareness given. As we narrow the context of the awareness, we render it more specific and adapted to the context.

Regardless of the local developer context, at the bottom right corner of the screen, the developer is always able to see a set of coloured squares, as shown in Figure 4.1.

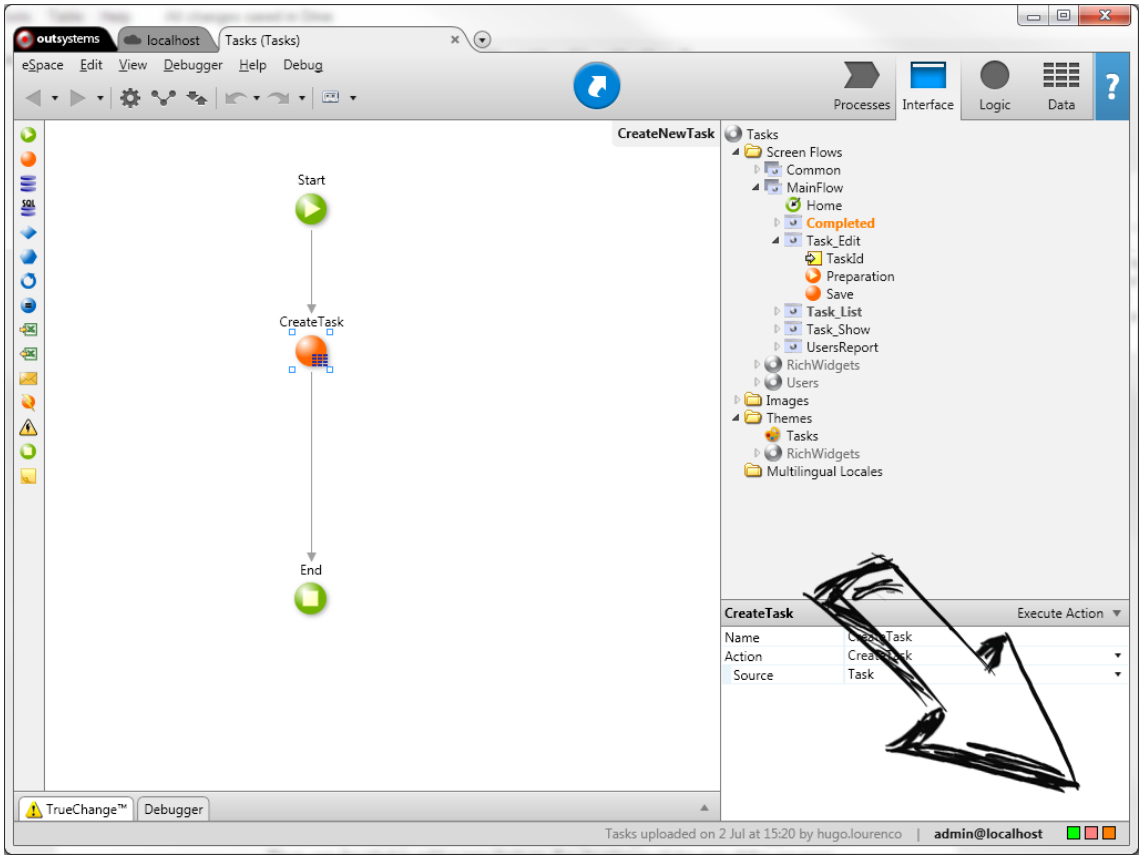


Figure 4.1: Service Studio: Global Awareness.

Each square represents a remote developer. Each developer is assigned a different colour, in each Service Studio instance, other than his own. In order to provide the developer with the relation between colour and username, along with current object in edition, the Service Studio gives a tooltip each time the local developer hovers over one of the squares. The tooltip shown is illustrated in Figure 4.2.

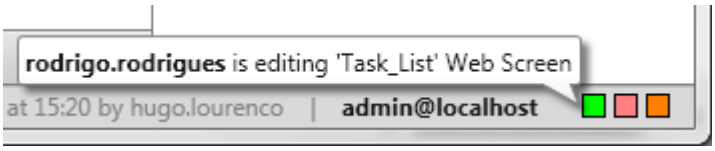


Figure 4.2: Service Studio: Global Awareness tooltip.

This type of awareness contributes to enable the developers to be aware of which other developers are, at the moment, online and editing the same eSpace has them. Furthermore, it enables developers to be aware of which objects are being edited by the remaining developers.

Occasionally, the name of the object might not be enough. For example, the local developer HL opened the eSpace in a state at which RR had recently created the "Completed" Web-Screen. HL did not know about "Completed" and does not know what is its purpose. However, HL is aware of the tasks of the project and can identify the purpose of a Web-Screen by looking at its content or appearance. Thus, we decided to add a new feature. If a developer clicks one of the squares, the whole Service Studio changes and takes the remote developer's vision, i.e. the same eSpace Tree tab is opened, the same object in the Content Editor, the same object is selected. This feature enables a rather swift inspection of what remote developers are editing.

4.1.2 eSpace Tree

Through the eSpace Tree, we were able to give a less but yet global information about the developers presence, since all the main components of an application (actions, DB tables and web-screens) are shown in the eSpace Tree, divided by tabs. The figure 4.3 illustrates the awareness given at the eSpace Tree Presenter.

An object whose name is bold and black coloured suggests that the object is being edited by the local user. In Figure 4.3, we can see that *Task_Show* web-screen is being edited by the local user. Furthermore, we can also see that *Completed* web-screen is coloured differently, apart from its font weight being bold. In this case, it means that some other developer (the developer identified by the orange

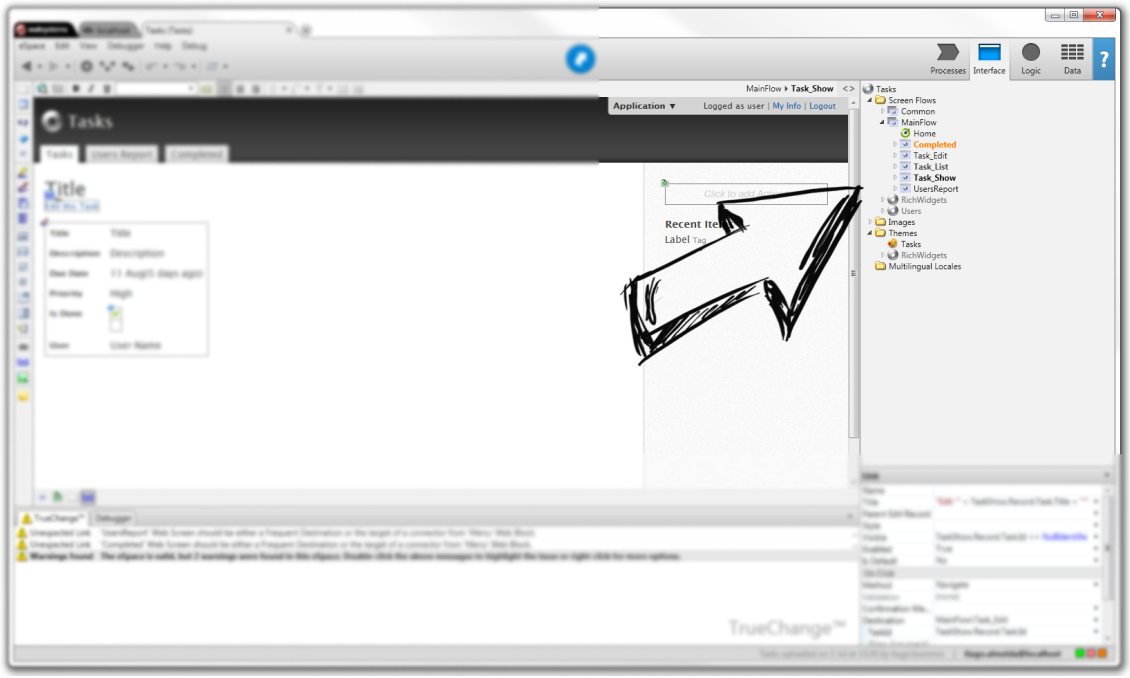


Figure 4.3: Service Studio: eSpace Tree Awareness.

colour) is editing the coloured object. The colour detail suggests that something is different but it may not be clear (to the local developer) what. Therefore, if one developer hovers over the object, Service Studio shows a tooltip providing the "explanation". This is shown in Figure 4.4.

When there is more than one remote developer editing the same object, e.g. *Task_List* web-screen in Figure 4.3, a default colour is used. This colour is only used in these cases. The given tooltip explains the reason why that object is coloured with a colour to which no user is assigned, as it can be seen in Figure 4.5.

Summarizing, the tooltip informs which remote developers are editing an object, if any, and the font colour of the objects' names is:

black if the local developer is editing it, or no one is;

multiple developers specific colour if there is more than one remote developer editing it;

user assigned colour otherwise;

Moreover, the font weight is bold if there is any developer editing the object, regardless of being the local developer or a remote one.

With this features, we were able to provide a quicker awareness inside the same eSpace tree tab. Assume that some developer wants to edit Completed Web-Screen. In order to check if another developer is already editing it, the local

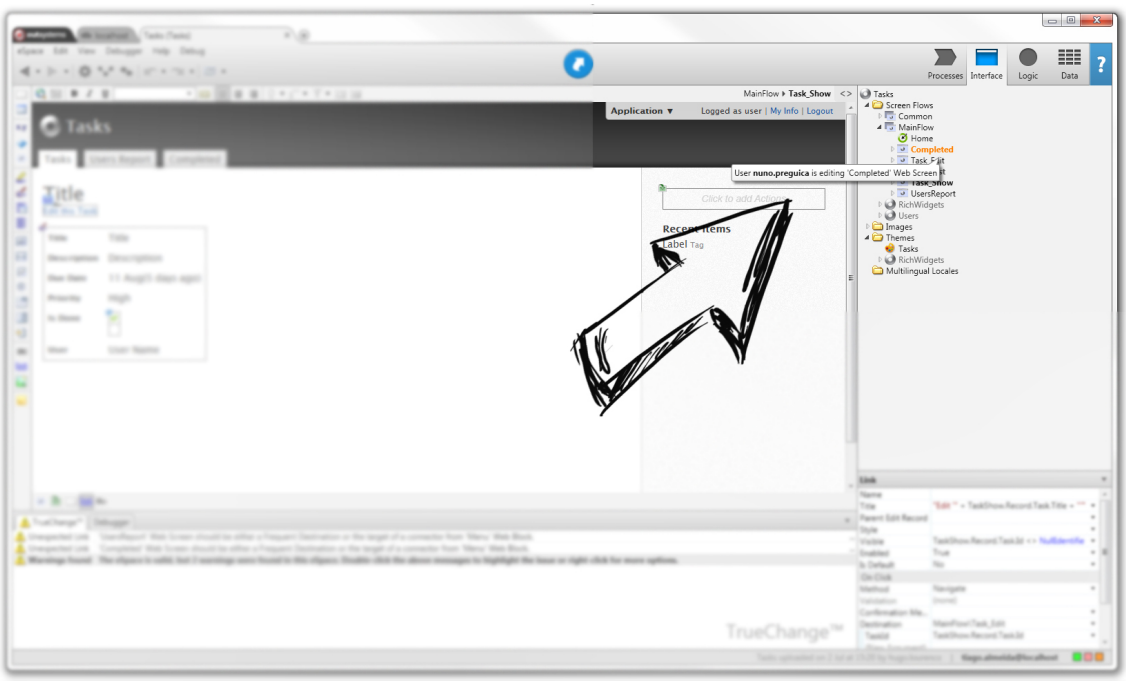


Figure 4.4: Service Studio: eSpace Tree Awareness tooltip.

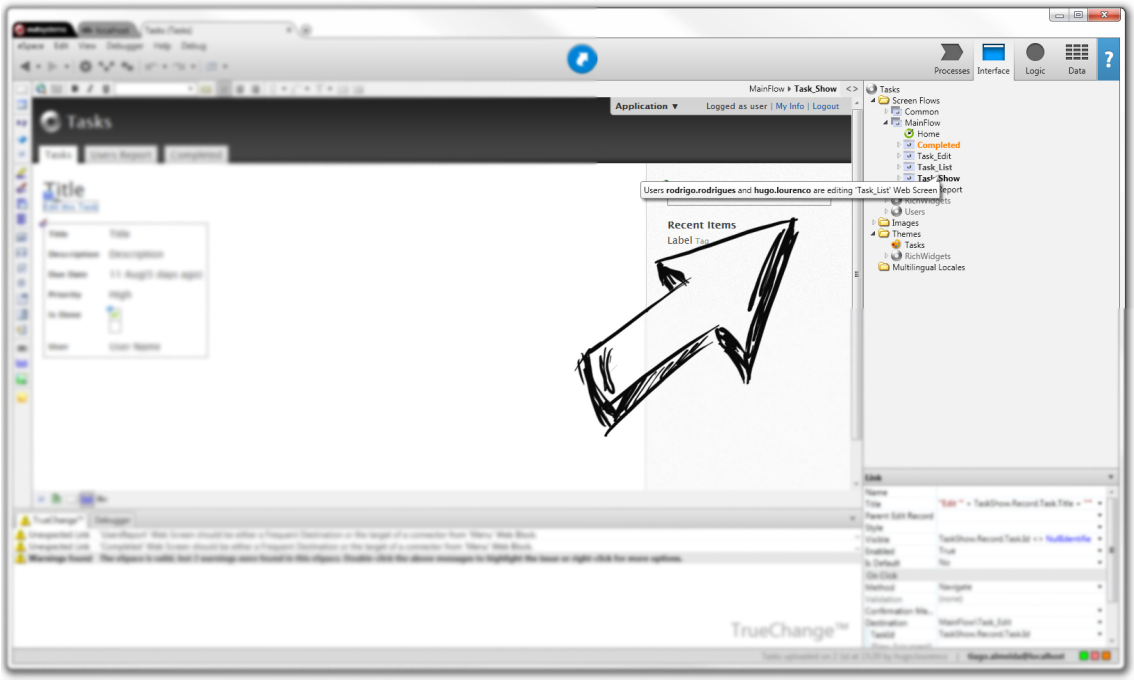


Figure 4.5: Service Studio: eSpace Tree Awareness tooltip.

developer is able to verify that simply by looking to the eSpace Tree Tab where the Completed Web-Screen is.

4.1.3 Content Editor

On the previous section, we explained that the font colour and weight of an object's description depended on the developers who were editing that object. However, we did not describe what would happen on one case: the case where the local developer and other remote developers are editing the same object. In these situations, we opted not to give any special type of awareness in the eSpace Tree, because we opted to give awareness inside the content of the object being edited. Figure 4.6 shows the Content Editor enriched with awareness details.

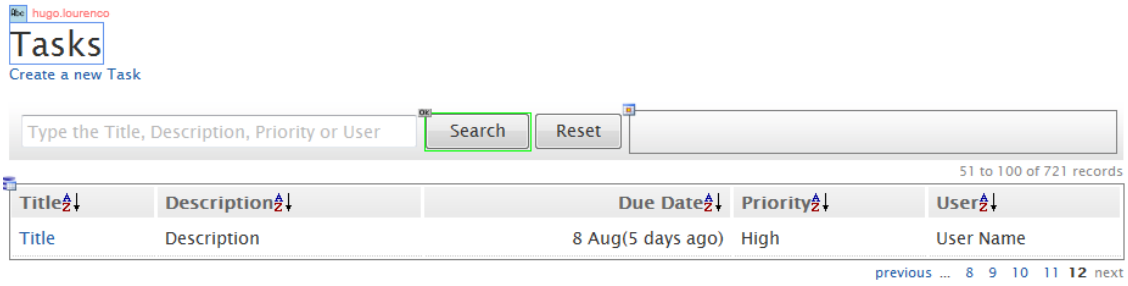


Figure 4.6: Service Studio: Content Editor Awareness.

The object adorned with a blue border is the object selected by the local user, which is the normal behaviour of the application. To inform the local developer about the presence of remote developers that are editing the same web-screen or user-action, we followed a similar approach. For example, the "Search" button in Figure 4.6 is adorned with a green border, which indicates that a remote developer - the one assigned to the green colour - has selected the object. Furthermore, if there is any remote developer that has selected the same object as the local developer, the names of such developers is presented on top of the object, as shown in Figure 4.6.

Rather than informing the local developer of which web-screens or actions some remote developer is editing, with this type of awareness, we are able to inform what are they editing inside it, e.g. the title of the page, the condition of a "if-then-else" node inside a user action, and more.

4.1.4 Properties Editor

Every object has a set of properties, which are only changeable through the Properties Editor. This is the main purpose of the Properties Editor. Moreover, specific details can be found there, such as the developer who created the object and the developer who last changed the object (see Figure 4.7). However, these informations were not updated in real-time, but only after the Merge Process occurred, because that would be the only time when a Service Studio instance would be aware of remote changes. After implementing the real-time collaboration, the several Service Studio instances are aware of remote changes immediately and that enables more frequent updates of these informations.

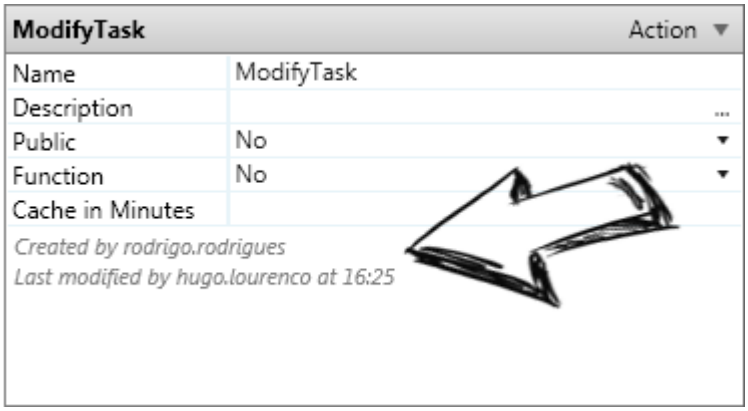


Figure 4.7: Service Studio: Properties Editor Awareness.

4.2 Implementation

The Service Studio implementation follows an Model-View-Presenter architecture. Moreover, every component (Content Editor, Properties Editor, among others) of the Service Studio is implemented by a different Presenter. These Presenters are organized in a tree architecture, briefly represented in Figure 4.8. Each Presenter is responsible for a set of details that are relevant, in this context. For example: for the Content Editor, the object in edition is relevant; for the eSpace-Tree, the opened tab is relevant; and so on.

Visual State Serialization

To provide the developers with awareness regarding other developers' presence, we need to serialize their visual state. By "visual state", we consider all needed details, such as: which object the developer is editing at the Content Editor, which

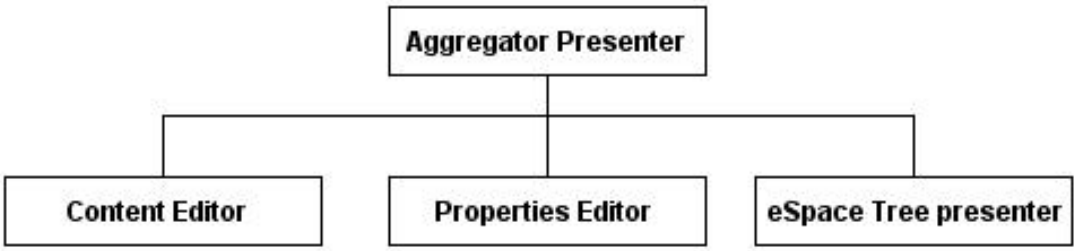


Figure 4.8: Service Studio: Presenters architecture.

object the developer has selected, which eSpace Tree tab is opened, among other. However, the OutSystems Platform already needed to save the visual state of a developer, for other purposes. Furthermore, the OutSystems Platform had already a solution.

Each Presenter has a private State, which abstracts all the above mentioned details. Moreover, the States follow the same tree-architecture as their Presenters. Furthermore, the Aggregator Presenter's State abstraction supports two operations, *SerializeToString* (State => String) and *Parse* (String => State), such that, for a State **s**, Parse(SerializeToString(s)) = s.

4.2.1 State Propagation

After serializing a visual state to a String, we have to propagate one developer's state to every other developer editing the same eSpace. In the previous chapter, we have explained the reasons why we opted to make use of Service Center as the mediator for all exchanged commands. For the same reasons, again, we opted to make use of the Service Center capabilities. Besides, the exchange of visual states is simpler than the exchange of commands, as it needs no sort of concurrency/conflict detection. The only requirement is to guarantee that information is delivered.

A Web-Method was developed, on Service Center, to enable the Service Studio instances to send their visual states and to check for the updated visual states of other Service Studio instances. Again, and for the same reasons as before, we opted for the polling approach. The Web-Method consists of the following signature:

Inputs

- eSpaceKey: String

- Instance_id: String
- UserName: String
- State: String

Outputs

- RemoteStates: List<Triplet<String, String, String»

The output list, RemoteStates, is a list of triplets consisting of a serialized visual state, the instance id of the instance corresponding to that visual state and, finally, the username of the developer logged at that instance (for presentation issues).

For the Service Center to maintain the states persistently across time, a database table was designed, which consists of the following attributes:

Instance_id the Service Studio identifier;

eSpace the eSpace identifier to which the state concerns;

UserName the username of the user logged on that instance;

State the serialization of the Presenter State;

In order to maximize the accuracy of the informations provided to the developers, each Service Studio instance polls the Service Center every 500 milliseconds.

4.2.2 State Processing

The component which is responsible for communicating with the Service Center is the Aggregator Presenter. The Aggregator Presenter is the root of the tree, which is the Presenters' architecture. Therefore, the algorithm to process the remote states begins at the Aggregator Presenter and follows a tree traversal behaviour, for each state.

The description in pseudocode:

```
void ProcessRemoteState(State state) {
  PassOnToChildrenStates(state);
  ProcessState(state);
}
```

```
void PassOnToChildren(State state) {
  foreach( childState in state.childPresenterStates) {
    var childPresenter = FindAdequateChildFor(childState)
    childPresenter.ProcessRemoteState(childState);
  }
}
```

Afterwards, each Presenter processes differently the remote state. For example, the eSpace Tree Presenter will process the section of the section of the State where it can find the object being edited by the remote developer. Other Presenters will not.

4.2.3 User Exit

On our implementation, we consider that awareness is only important when regarding the developers that are actually online and editing. To detect when a given developer is online or not, the Service Center resorts on the HeartBeats database table, mentioned in the previous chapter. The usage is similar. Thus, when a Service Studio instance polls the Service Center in order to obtain the remote instances' visual states, the Service Center only sends the states of those instance whose last heartbeat is recent. By recent, we mean that the last heartbeat happened less than 30 seconds ago.

The Service Studio keeps track of which instances are online. Thus, when a remote developer goes offline, the Service Studio is able to detect and remove all awareness details regarding that remote developer, since the visual state of that remote developer is no longer present in the Service Center responses.

4.3 Optimizations

In this section, we will present some optimizations, in order to reuse code and eliminate part of the communication overhead.

Unnecessary Propagations

During a session of some developer, there are periods of time during which the visual state of the developer might not change. For example, while the developer is thinking about what to do or while the developer is writing a SQL query,

among other activities that do not change the developer's visual state. During such periods, the Service Studio keeps on sending the local visual state to the Service Center. However, this is unnecessary.

It seems reasonable, in those periods, not to send the local visual state to the Service Center. However, if the visual state does not change during several seconds and the Service Studio does not send it during that time, the Service Center perceives it as if that Service Studio instance is offline.

To avoid both sending unnecessary information and confusing the Service Center, we opted to keep communicating with the Service Center but, instead of sending the serialized visual state, the Service Studio sends an empty string. When the Service Center receives an empty string, it perceives that the instance is still online, but the visual state remains the same. Thus, the Service Center updates the Last HeartBeat of that instance and preserves the State stored in the database table.

On the contrary way of the communication, the Service Center needs not to send a state to a Service Studio instance that already has received that same state. Again, not sending a state from one Service Studio instance to another will lead the latter instance to perceive that the former is offline and, consequently, remove all visual details regarding that instance. Thus, when the State of an instance is unchanged, the Service Center sends an empty string instead of the serialized visual state. The Service Studio instance, upon receival of an empty string as a visual state, perceives that there were no changes and preserves the visual details.

Single Web Method

Up to this point, we have two web-methods. One for sending and getting commands and, another, to send and receive visual states. Each of the operations is repeated every 500 milliseconds. Furthermore, these operations share some common inputs: the Instance_id, UserName and ESpaceKey. Moreover, these operations are both executed by the Aggregator Presenter.

Merging these two operations in a single operation is harmless and avoids problems with concurrency. For example, suppose that the developer John created an action, named "Action1" with key "x+q0", and consequently, opened it for edition. Now, suppose that John's Service Studio instance sends, separately, the command and the visual state after the creation of the action. A few milliseconds later, Bob's Service Studio instance, concurrently, polls the Service Center (separately) both for commands and for visual states. What Bob's instance obtains is a command, *AddUserAction*, and a state which indicates that Bob's editing

the Action identified by the key "x+q0". If the processing of the visual state is faster than the execution of the command, the Service Studio will try to present visual awareness to Bob on an object that does not exist yet. In that case, there are some possible scenarios: it causes an error; Service Studio prevents the error by ignoring that detail; Service Studio postpones the processing of that visual detail. Either way, having two closely-couple operations being called separately and concurrently, raises a few issues. Therefore, we opted to merge these two operations in a single operation, which has the following inputs and outputs:

Inputs

- Instance_id: String
- eSpaceKey: String
- UserName: String
- State: String
- Commands: List<String>

Outputs

- RemoteStates: List<Triplet<String, String, String»
- RemoteCommands: List<Pair<Integer, String»
- CommandsOrder: List<Integer>

Thus, upon reply from the Service Center, Service Studio processes commands and, afterwards, it processes the remote visual states. This prevents issues raised by concurrency and simplifies the implementation, since both tasks were performed by the Aggregator Presenter.

4.4 Limitations and Future Work

The visualizations provided simply skim the problem of how to enhance eSpace awareness in a development team, by informing developers (in real time) about the presence of other developers editing the same eSpace. The solution has a few limitations. However, there are some topics where future work is possible. In this section, we will discuss these topics.

4.4.1 Visual details provided

The visualizations implemented support most sections of an eSpace. However, due to time limitations for the prototype development, we did not implement visualizations in every section of an eSpace. The Content Editor is the main section of edition of an eSpace. On the Content Editor, one can edit several different types of objects. For our prototype, we opted to provide awareness when editing web-screens. We intend to expand and bring awareness to all sections of Service Studio.

4.4.2 Visualization and Colour Scheme

The visualizations we provide to enhance eSpace awareness are mostly based on enhancing objects by colouring them. The colour palette we chose is limited and does not scale for more than 11 developers in the same eSpace. Moreover, the colours picked were simply selected by their contrast to black and by the fact that they draw the attention of people, without thorough analysis. As future work, we should thoroughly study and choose the appropriate colours.

Other relevant issue that one must consider: how to provide developers' presence awareness to colour blind people. Colour blindness occurs in 8% of males (10 times oftener than females) [Whi83], which indicate that this is an issue to consider. In this case, we should provide visualization details other than just changing the font colour of an object's display name. One possible approach, for example, on the eSpace-Tree, would be to underline the object's display name with an accordingly coloured line. This way, a common developer would perceive which developer is editing an object as before. On the other hand, a colour blind developer would understand that some developer is editing that object and, if we wanted to know who, he just had to hover over the object to get a tooltip. Regarding global awareness, instead of having simple coloured boxes, we could enrich the squares with the initials of the corresponding developers inside the boxes.

5

Conclusions and Future Work

Software development in mainly a team-oriented activity. The software development has been mainly performed in a collocated setting, where team elements share the workplace. However, recently, this scenario has begun to change and software development becomes a global activity, where team elements are dispersed across distant places.

Collaborative software development comes with several challenges such as coordination and communication between team elements, in order to produce quality software. Conflicts occur when the work efforts of several elements collide in some shared resources. These challenges increase with the physical distance between the team elements.

In our thesis, we focus on the improvement of collaboration within software development teams, especially those physically dispersed. Our thesis is that by making changes visible to all team elements, as they occur, reduces the risk of conflict and, therefore, reduces the time wasted solving them. To validate this claim, we presented a solution that represents the evolution of web applications as sequences of fine-grained changes (commands).

Furthermore, we believe that providing visual details regarding developers' presence and actions will help improve the team awareness.

5.1 Future Work

During the development of the proof of concept, we used the existing command serialization. Those commands need a particular presenter to be opened and, therefore, cause a flickering on the screen, i.e. quick changes on the screen. As it was mentioned before, we should address this issue and design a new command serialization, such that the commands become self-sufficient, enabling the *Event Playbacker* to reproduce commands directly on the model, without communication with the Presenter.

Usability Tests

Due to the limitations posed by the command serialization, we were not able to realize usability tests to the proof of concept. The reason is that the flickering harms the user experience and becomes difficult to work, especially during periods of high activity. Moreover, during the development of a given task, the developer is most likely to edit more than web-screens on the Content Editor. Therefore, after redesigning the command serialization and providing user-presence awareness on all sections of the Service Studio, we intend to perform Usability Tests.

The Usability Tests we want to perform will be composed by a set of tasks. We will have two, or more, developers performing a set of tasks simultaneously. The initial tasks will have a greater distance between the developers than the last ones. In other words, during the initial tasks, developers will edit disjoint sections of the eSpace. As they complete those tasks, their sections become closer to the point that they will be working on the same section, e.g. web-screen.

We intend to analyse how fast they notice visual details provided to enhance awareness and how they interpret them. Furthermore, we intend to analyse how fast they notice the fact that they are collaborating in a real-time manner. Moreover, we want to analyse how they feel about the lack of isolation, which characterizes the real-time collaboration, and how they feel about not having the need to merge work after a session of development. Finally, we want to test the productivity developers achieve with the real-time approach.

Correctness Tests

Every software application that goes into production must first go through a battery of tests. The tests this application would have to go through, if it was to go

into production, would be tests to the correctness of the product.

In order to test the Service Studio behaviour, one could install a Mock Server that would give pre-designed replies to Service Studio. For example, when the tester-user creates an action and the Service Center propagates it to the server, the Mock Server replies with a few more actions, having the action created by the user being in the middle of the others (considering the order assigned by Service Center). This way, we would make the Service Studio undo the last command, execute some remote commands, re-execute its command and, finally, execute some more commands. Then, test if the keys match the actions, i.e. every created action has the Object Key it was suppose to have. Furthermore, guarantee that, for example, *Action1* does not have the Key that *Action2* was supposed to have.

To test the Service Center, we could develop a thread that would play several "Service Studio" roles and make each Service Studio send commands and check if the Service Center is replying with the commands and command orders it was supposed to.

Bibliography

[Cam02] J.D. Campbell. Multi-user collaborative visual program development. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 122–130. IEEE, 2002.

[Ced93] Per Cederqvist. Version management with cvs, 1993.

[CW00] A. Cockburn and L. Williams. The costs and benefits of pair programming. *Extreme programming examined*, pages 223–247, 2000.

[DB92] Paul Dourish and Victoria Bellotti. Awareness and Coordination in Shared Workspaces. (November):107–114, 1992.

[DISK07] D Damian, L Izquierdo, J Singer, and I Kwan. Awareness in the Wild: Why Communication Breakdowns Occur. *International Conference on Global Software Engineering ICGSE 2007*, 6:81–90, 2007.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM.

[GST05] Tom Gross, Chris Stary, and Alex Totter. User-centered awareness in computer-supported cooperative work-systems: Structured embedding of findings from social sciences. *International Journal of Human-Computer Interaction*, 18(3):323–360, 2005.

[IDE] Cloud9 IDE. https://c9.io/.

[IMO+03] A. Imine, P. Molli, G. Oster, M. Rusinowitch, et al. Proving correctness of transformation functions in real-time groupware. *Proceedings of the*

eighth conference on European Conference on Computer Supported Cooperative Work, pages 277–293, 2003.

[IN04] Claudia-Lavinia Ignat and Moira C. Norrie. Grouping in collaborative graphical editors. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 447–456, New York, NY, USA, 2004. ACM.

[MFV] M. Mateus, P. Ferreira, and L. Veiga. Vector-field consistency for collaborative software development.

[PMSL09] N. Preguica, J.M. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 395 –403, june 2009.

[PPR+83] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, May 1983.

[PR11] Chris Parnin and Spencer Rugaber. Resumption strategies for interrupted programming tasks. *Software Quality Journal*, 19:5–34, 2011.

[SC02] Chengzheng Sun and David Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Trans. Comput.-Hum. Interact.*, 9(1):1–41, March 2002.

[SJZ+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distrib. Comput.*, 7(3):149–174, March 1994.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin Heidelberg, 2011.

[Sub] Apache Subversion. http://subversion.apache.org/.

[SVF07] N. Santos, L. Veiga, and P. Ferreira. Vector-field consistency for ad-hoc gaming. *Middleware 2007*, pages 80–100, 2007.

[SW02] David Stotts and Laurie Williams. Distributed pair programming. In Don Wells and Laurie Williams, editors, *Extreme Programming and Agile Methods — XP/Agile Universe 2002*, volume 2418 of *Lecture Notes in Computer Science*, pages 283–283. Springer Berlin Heidelberg, 2002.

[Whi83] M G Whillans. Colour-blind drivers' perception of traffic signals. *Canadian Medical Association Journal*, 128:1187–1189, 1983.

[WK03] L. Williams and R.R. Kessler. *Pair programming illuminated*. Addison-Wesley Professional, 2003.