Giuliano Giorgio Ragusa

Bachelor in Computer Science

Code Reviews for Visual Programming Languages

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

Master of Science in
Computer Science and Informatics Engineering

Adviser: Miguel Carlos Pacheco Afonso Goulão,
Assistant Professor, Faculdade de Ciencias e
Tecnologia da Universidade Nova de Lisboa

Co-adviser: Henrique Gabriel Henriques,
Software Engineer, Outsystems

Examination Committee

Chairperson: Doutor António Maria L. C. Alarcão Ravara
Raporteur: Doutor João Carlos Pascoal de Faria

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2018

Code Reviews for Visual Programming Languages

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

Acknowledgements

I choose this to be my last piece of writing on this journey so I could be sure of including all the ones that helped me all the way to the end.

I want to start by saying I am grateful to my coordinator Miguel Goulão, who has provided me with guidance every time I have needed. OutSystems for giving me the opportunity of meeting and sharing the work-space with amazing people for the last year. The back-end team for all the coffee times, team lunches, episodes and more important for letting me win the races. Not forgetting Henrique, who I want to especially thank, for being my Guru during the last year and for the unfailing support along the way.

Also, I would like to thank all my desk mates which made the thesis team the best out there. Firstly, to Mariana for her immeasurable dedication to both our thesis, warning me of deadlines, being my on-the-fly tester, graphic designer, driver for the events, but most of all for being a really true friend. Then, to junior, Miguel, for not giving up on our snooker battles, where he stood no chance.

Further, I want to acknowledge all my friends for supporting me, without even understanding the subject of my thesis, independently of how many times I tried to explain.

Finally, I must express my profound gratitude to my family for providing me with unconditional support and constant encouragement throughout my years of study. Lastly but not least to Beatriz for being there 24/7 and giving me the mental strength I needed to finish this journey.

This accomplishment would not have been possible without you. Thank you to all.

Abstract

Visual languages such as *OutSystems* are being progressively more used by large teams that collaborate on enterprise grade projects. This results in new challenges when it comes to ensuring the quality of the applications built with this languages, without sacrificing productivity. An increasingly common way to improve quality is performing distributed lightweight tool supported code reviews.

A code review consists of an analysis of the code by one or more developer. The objective for this is to ensure that the changes do not introduce bugs and adhere to the established coding best practices. With this practice, companies can offer higher quality and easier to maintain software. However, the current tools and processes for code review are focused on textual programming languages and opening a bigger gap for the visual ones.

The main goal of this dissertation is to develop a specialized tool to promote and making visual programming languages code review more accessible. An easy-to-follow process was also created to support the tool. At the moment, the tool can be used with *OutSystems'* language and UML class diagram, however, new languages can easily be added to the solution.

We examined the code review method currently used by developers vs. the new method created concerning defect detection rates and user experience. Also conducted individual interviews with 30 computer science engineers. Results were collected with the help of questionnaires and analyzed using multiple descriptive statistics. The new code review method was significantly better than the current method used, reaching average usability improvements of 47.75%, tasks load reduction of 15,43%. Generalization of results is limited since the analyzed techniques were applied only to a small set, consisted of 30 testers.

The result supported the evidence concerning the advantage of the new code review method using the developed tool over the current method of reviewing visual artifacts.

Keywords: Code Review, Visual Programming Languages, Inspection processes.

Resumo

Linguagens visuais como a da *OutSystems* têm sido cada vez mais utilizadas por grandes equipas, que colaboram em projetos a nível empresarial. Como resultado, surgem novos desafios quando se trata de garantir a qualidade das aplicações criadas com estas linguagens, sem sacrificar a produtividade. Uma maneira cada vez mais comum de melhorar a qualidade está em realizar revisões de código suportadas por ferramentas ligadas e distribuídas.

A revisão de código consiste numa análise efetuada por um ou mais programadores. O seu objetivo é garantir que as alterações efetuadas não afetam o código fonte, não introduzem defeitos e que respeitam as regras de boas práticas de programação. Com esta prática, as empresas conseguem melhor a qualidade do *software*, bem como a sua manutenção. Contudo, as ferramentas e processos de revisão de código que existem atualmente, focam-se em linguagens de programação textuais abrindo uma grande lacuna no que às linguagens visuais diz respeito.

O principal objetivo desta dissertação consiste em desenvolver uma ferramenta especializada para promover e tornar as revisões de código em linguagens de programação visuais, mais acessíveis. Além disso, foi criado um processo simples para suportar a ferramenta. Até ao momento, a ferramenta apenas pode ser usada com linguagem *OutSystems* e com diagramas de classe UML, no entanto, novas linguagens podem ser facilmente adicionadas à solução.

Os métodos de revisão de código atualmente usados pelos programadores de linguagens visuais foram comparados com o novo método criado, relativamente ao fator usabilidade e taxa de esforço. Foram individualmente inquiridos 30 engenheiros informáticos com e sem experiência previa em revisões. Os resultados foram recolhidos através de questionários e analisados usando estatísticas descritivas. Ficou provado que o novo método de revisão de código apresenta vantagens significativas comparativamente com o método atualmente usado, alcançando melhorias na usabilidade de 47.75% e redução da taxa de esforço na ordem dos 15,43%. Contudo, a generalização dos resultados é limitada, uma vez que as técnicas de análise foram aplicadas a uma pequena amostra.

Os resultados alcançados fortaleceram evidencia no que diz respeito as vantagens do novo método de revisão comparativamente com o método atual.

Palavras-chave: Revisão de Código, Linguagens de Programação Visuais, Processos de Inspeções

Contents

Chapter 1

Introduction

This chapter contains the introduction of the dissertation. Starting by presenting a description of its focus and context, their motivations. Then, the objectives and contributions are introduced. The chapter ends with a presentation of the document's structure.

1.1 Context and description

Visual programming languages (VPLs) are any programming languages that allows users to create programs by manipulating software elements, graphically instead of having to specify them textually. While a typical text based programming language makes the developer think like a computer, a VPL lets the developer describe the process in terms that make sense to the human [1][2]. Even though no code platforms[1] have already demonstrated to be effective in various environments [3], they have limited capabilities. There is a whole set of similar tools that take the best of visual programming and combine it with text based coding. These new tools are denominated low code [4].

OutSystems Platform is one of these tools. Using the Integrated development environment (IDE) created by OutSystems called Service Studio (more on OutSystems and Service Studio in Section 2.1), developers can create software visually by drawing interaction flows, UIs and the relationships between objects, but also with the capability of using textual code if they feel it is a better alternative.

This fusion between the VPLs and text-based programming fits well the modern need's of software development. Low code tools help to decrease the complexity of software development and create a world where a single developer can create rich and complex systems, without having to learn all the underlying technologies [4].

One of the purposes of this dissertation is to understand how code review is handled when the object of the review is specified using VPLs which type of processes

[1] **No code platforms** allow developers to create web applications using point-and-click/drag-and-drop, metadata model methodology

and tools are available to ease this process. Finally, using the gathered data, develop a tool and a process for code reviewing visual languages. In order to achieve this, the OutSystems platform was used as a case study.

1.2 Motivation

Book authors need editors to identify errors. It is human nature that an author cannot adequately proof read his own work [5]. Authors of software need the same assistance as authors of novels to achieve the goals of the software organization. In the software development world, this is called code review [5]. It's importance has been acknowledged for a long time. In 1986, in an interview [6], when asked what the best way to prepare to be a programmer was, Bill Gates answers:

> *"You've got to be willing to read other people's code, and then write your own, then have other people review your code. You've got to want to be in this incredible feedback loop where you get the world-class people to tell you what you're doing wrong ..."*

A code reviewer does not only look for bugs, but at the same time, they are constantly exposed to new ideas and other developers knowledge, making them write a better code. Code review facilitates the communication of institutional knowledge between developers. Experienced team members have the opportunity to impart their wisdom and advice; this also encourages team bonding. It is at the same time, a solution to ensure maintainability. Through code organization and adequate comments, code review enables a person uninvolved in the project to read a portion of the code, understand what it does and make changes if necessary.

However, despite the benefits of software inspections in general, VPLs have so far not been adequately addressed by inspection methods. This represents a problem for two reasons: First, over the last decades, these model development methods have replaced conventional structured methods as the embodiment of "goodness"in software development, and are now the approach of choice in most new software development projects [7]. Techniques limited to conventional structured methods are expected to become more and more irrelevant for the development of new software products as these methods are superseded. Secondly, despite its many beneficial features, low defect density is not one of the strong points of the VPLs paradigm.

Visual programming languages would therefore, profit enormously from the availability of advanced and experimentally validated defect detection techniques.

To our knowledge, experimental work or analysis has not been done on how to inspect these type of visual languages. The problem of code review continues because no solution has been found for these languages.

1.3 Objectives

This dissertation primary objective is to develop a tool and a process to help developers to review peers' visual code. Besides, we want to analyze the current code review methods available for visual languages against the new method created. The methods are evaluated concerning usability and the tasks load rates. The evaluation includes

interviews and usability tests conducted from the viewpoint of potential users in the context of academic and enterprise usage.

1.4 Expected key contributions

This dissertation main contribution is a code review tool for VPLs. Due to the evolution of the programming languages, the technologies involved also need to evolve accordingly, including code review. Existing tools are defined based on textual languages, but not all can be directly used on VPLs. Therefore, this dissertation, inspired on the existing mechanisms and tools available for textual languages code reviews will focus on, creating an innovative and effective code review tool dedicated to VPLs.

Besides, an easy-to-follow process is defined to support the tool users. The process guides participants over the entire code review process explains how to use the tool and proposes a process work flow.

1.5 Structure

This document is organized, excluding the current chapter, in the following way:

- Chapter 2 - Background: includes an overview of the OutSystems Platform and its architecture. Also, offers a brief explanation of the code review concept, describing different techniques for code reviewing ;
- Chapter 3 - Related work: a brief overview of the methods and tools currently available;
- Chapter 4 - Requirements elicitation: a research of the requirements of the stakeholders;
- Chapter 5 - Proposed solution: presentation of the tool and the process for visual programming languages code review;
- Chapter 6 - Results: an analysis of the results obtained through usability experiments of the original and new method and possible threats to their validity;
- Chapter 7 - Conclusions and future work: an overview of what the dissertation achieved and suggestions for future work.

Chapter 2

Background

In this chapter, we introduce OutSystems, with a description of its purpose and an overview of the platform's structure. Besides, we explain the origin of code reviews, starting from the code inspection process up to the concept of lightweight code review. Finally, we describe some of the techniques used for defect detection in software inspections as well as the method in other major software development companies.

2.1 OutSystems

OutSystems is a software development company that offers a high productivity solution using a low code development platform. Said platform, enables visual development of an entire application both web and mobile, allowing a high level of abstraction by not having to worry about low level details related to developing and publishing applications. As a result, faster and increased quality result can be achieved in comparison to general purpose languages [8].

2.1.1 OutSystems platform

The OutSystems platform [9], shown in Figure 2.1, is composed by three main components: Service Studio, Platform Server and Application Server. OutSystems also have an a separate platform tool, where the users can integrate extensions to the low code platform.

2.1.1.1 Service studio

Service Studio is the development environment (See figureFigure 2.2) provided by the *Outsystems Platform*. It allows applications to be developed in modules called *eSpaces* that are later reused by other applications. These modules contain process definitions, user interfaces, business logic, and the application's data models. By drag-and-dropping, developers can compose all the components necessary to define web or mobile applications completely.

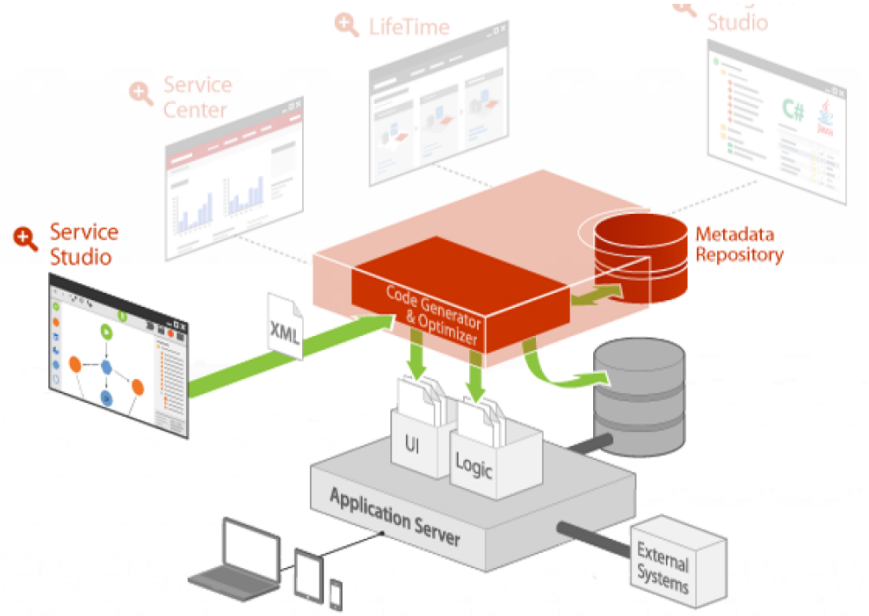


Figure 2.1: *OutSystems* Platform architecture [10]

The IDE has four main views, which enable the developer to create user interfaces, business processes, databases and define custom logic as desired. The visual application models created using *Service Studio* also have a textual representation in Extensible markup language (XML) [11].

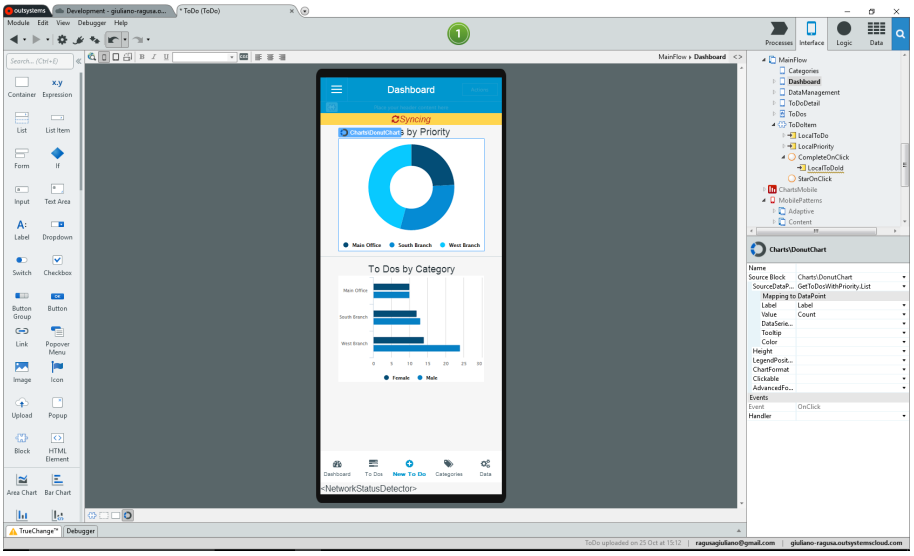


Figure 2.2: Screenshot of *Service Studio* environment

2.1.1.2 Platform server

The Platform Server uses the application model to generate code that depends on the particular stack being used, e.g. for a Windows Server [12] using SQL Server [13] this

will be ASP.Net [14] and SQL code. Once this process has been finalized, the compiled application is then deployed to the Application Server.

2.1.1.3 Application server

The Application Server runs on top of Docker Containers [15] and IIS [16]. The server then stores and runs the developed application which is connected to a relational database management system, which can be SQL Server, Oracle [17] or MySQL [18]. The SQL code generated by the Platform Server is specific to the selected database management system.

2.1.1.4 Integration studio

Integration Studio is the Agile Platform that allows to create and manage extensions.

This tool creates the connect with entities from other databases and also to define functions whose operation is determined in low level languages such as Java or .NET code.

2.2 Code review

The motivation for software inspections has emerged as a way to detect and remove defects before they propagate to subsequent development phases where the detection and removal become more costly for their stakeholder [7].

The first structured approach appeared in 1972 when Michael Fagan defined the inspection process [19]. Such process had two main goals: find and fix product defects, find and fix development process defects that lead to product defects.

Figure 2.3 outlines the Fagan Inspection process. A Fagan Inspection is composed of 6 steps:

1. **Planning.** The moderator checks that materials meet the entry criteria; arrange the availability of the participants; arranges the place and time of the meeting.

2. **Overview.** Group education of participants on the materials under review; the moderator assigns the roles to the participants.

3. **Preparation.** The reviewers go through the material individually with the goal of understanding it thoroughly.

4. **Meeting.** The inspection team assembles. The reviewers express their findings, and the recorder records them.

5. **Rework.** The author reassesses and modifies the work product. The rework is verified by holding another inspection.

6. **Follow-up.** The moderator assures that all defects have been fixed; assure that no other defect has been introduced.

Studies have shown that somewhere between 50% and 90% of the defects in software artifacts are detected and corrected through inspections [20].

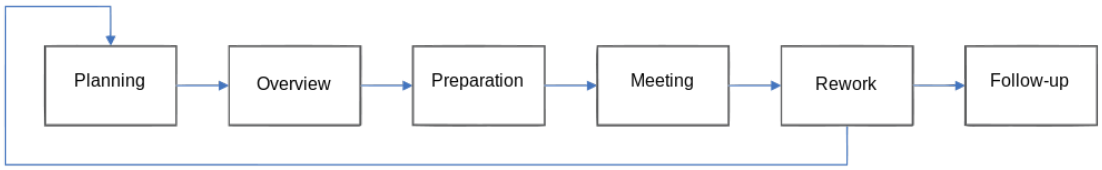


Figure 2.3: Fagan Inspection process

Since the introduction of code inspections, new studies and experiments to assess the use of the inspection process have emerged. All of which, are used to propose improvements over the inspection process that will finally contribute to the development of higher quality code while also lowering costs and development time.

2.2.1 Type of review

Code reviews are divided in two different types based on when they are performed. The two different approaches are pre-commit and post-commit [21], an example of these can be found in section 3.1.

2.2.1.1 Pre-commit

Pre-commit allows for checking the quality of the changes before they are applied: this allows developers to assess that the changes in the code satisfy the project standards and that bugs are not introduced in the code. The down-side of this method is that it increases the release period since the change has to be first approved. This means that developers are not able to work with the changes for a longer time, possibly slowing down the development time [22].

2.2.1.2 Post-commit

Post-commit, on the other hand, immediately applies the changes, then the code is reviewed. This approach enables developers to keep working on the code while waiting for the review to be completed, allowing for a faster release cycle. The downside of this approach is that it is more likely to have bugs introduced in code and that there is no guarantee that the review will ever take place [22].

2.2.2 Benefits of code review

It has been reported in many researchers' work that inspections are an effective method of detecting defects which allows a significant rise in productivity, quality and project stability [20]. Travassos is one of the authors of these researches [23]. In his work, he adapts Wheeler [24] report and presents clear evidence of these facts, as below:

- AT&T raised productivity and quality by 14%, being inspections 20 times more efficient than testing.
- IBM obtained a 23% raise in code productivity and 38% reduction of defects found in testing stage.

- Fagan's work reported that inspections detected 93% of all defects in a program at IBM. T he effectiveness of inspections was also over 50% in two other projects [20].
- HP typically found 60% to 70% of the defects using code inspections [25].
- Inspections used in many occurrences indicated that the effectiveness of code inspections were typically in the range of 30% to 75% [26].

Since code inspections are human-based activity, their cost is valued by the human effort. In the literature, one of the most addressed questions is whether the inspections' effort is worth compared to other quality assurance activities, such as testing. Many studies have presented solid data supporting that the costs for detecting and removing defects during inspections are much lower than detecting and removing the same defects in subsequent phases [7].

Some of these studies have shown that:

- The ratio of the cost of fixing defects during inspection to fixing them during formal testing range from 1:10 to 1:34 [27].
- The time needed per defect varies depending on the development process. However, in Weller's report [28], he states that the time needed per defect in a code inspection is 1.43 hours per defect in inspection and 6 hours in testing.
- Kelly [29] found that, on average, the time needed to detect and fix a defect is 1.46 hours in code inspections; and 17 hours per defect in testing.

Conclusion, both inspections and testing are very important in detecting defects. However, correcting defect found in later developmental phases has a much higher cost. At the same time, considering effectiveness, inspections also have higher rates when compared to testing.

2.2.3 Modern code review

In recent years, the code review process has become more lightweight compared to the code inspection created by Fagan. This lightweight process has been named Modern Code Review by Bacchelli and Bird [30], to differentiate between the old process of inspection described by Fagan from the process that has been recently evolving. This process is being used by many companies [31], and it is characterized by being informal and tool-based. Through its basis in tools, modern code review has the advantage of being asynchronous, whereas with previous software inspection processes would be cumbersome. These web-based tools allow teams spread around the globe to continue their work without the problem of scheduling meetings.

2.3 Code review techniques

To maximize their potential, these reviews must be as thorough and detailed as possible. This reflects the need for systematic techniques that tell participants what to look for and more importantly, how to scrutinize a software artifact.

In this section, we present an overview of existing methods and techniques for software inspections.

2.3.1 Reading techniques for defect detection in software inspection

A reading technique gives instructions to the inspector to guide the inspection process [7]. Given the inspection process importance, adequate support for inspectors during defect detection can likely result in dramatically raising the inspection effectiveness and efficiency. Reading techniques can vary a lot between each other. However, all the variations have a common goal which is checking whether a document satisfies quality requirement such as, correctness, consistency, testability or maintainability [7]. The most popular reading techniques are *adhoc* reading and checklist based reading [32]. Even though these are the most used, it does not mean they are considered to be the best. However, they require a lot less training. Alternative reading techniques also exist and will be briefly explained.

2.3.1.1 Adhoc

Adhoc is a Latin phrase meaning "for this". In other words, it refers to something made or happening only for a particular purpose or need, not planned before it happens [33]. Accordingly, *adhoc* reading offers no or very little guidance to the inspector, and the software artifact is simply given to inspectors without any specific direction or guidelines on how to review the software and what to look for [34]. Hence, inspectors must use their own intuition and expertise to determine how to go about finding defects in a software document. This makes *adhoc* reading very dependent on the skill, the knowledge, and the experience of the inspector. However, training sessions in program comprehension before the take off of inspection may help the inspector create some of these skills to lighten the absence of specialized reading techniques [35].

2.3.1.2 Checklist-based reading

In contrast to *adhoc*, in Checklist based reading (CBR), the inspector has to go through the artifact and answer a list of questions given. The questions are written to attract the attention of the inspector to specific aspects regularly defective on that kind of artifact. The checklists help the reviewers by giving them support. This help makes the review not as dependent on the experience of the individual reviewers as in *adhoc* reading. The software inspection checklists can differ from organization to organization or even within a project (See example in Figure 2.4) [36]. Therefore, checklists are seen as specific rules which may be, for example, company quality standards.

Although CBR approach is more supported than *adhoc*, it also has some disadvantages [37]:

- only particular types of defects are detected (mainly the ones covered by the checklist);
- inspectors frequently miss difficult to find defects because they do not have a profound knowledge of the artifact;

- defects and defect types that were not previously detected can be missed since the inspectors pay attention to the classes of defects present in the checklist;
- some of the questions might not be appropriate for the artifact; questions can be very general;
- there is typically no guidance on how to answers the questions in the checklist (i.e., should the inspector read the whole document first and then answer the questions, or review the document while contemplating a single question at a time);
- the reviewers have to go through excessive amounts of information contained in the checklists.

There are three basic criteria for evaluating the quality of UML-design diagrams: Correctness, Completeness, and Consistency. Each one of these criteria can be operationalized as follows:

Correctness
A diagram or a set of diagrams is correct if it is judged to be equivalent to some reference standard that is assumed to be an infallible source of truth. In the case of design diagrams, the analysis diagrams can be considered as such.

Completeness
A diagram or a set of diagrams is complete if no required elements are missing. It is judged by determining if the entities in the (set of) diagrams describe the aspects of knowledge being diagramed in sufficient detail for the part of the system under development.

Consistency
A diagram or a set of diagrams is consistent if there are no contradictions among its elements. This may be judged by checking the consistency between the elements of a single diagram or between several diagrams/ diagram types

Item no.	Where to look	How to detect
1	**All Design Diagrams**	Is each name unique?
2		Are all names used in the diagrams consistent?
3		Are all names used in the diagrams correct?
4	**Collaboration Diagrams**	Are the parameters and parameter types correct when compared to the “reads” and “changes” clause in the operation schemata?

Figure 2.4: Example of the checklist’s questions [7]

2.3.1.3 Perspective based reading

Perspective based reading (PBR) [38] another reading technique used in software inspections. The purpose of perspective-based reading is to inspect a software artifact and to identify defects from the perspectives of the software’s stakeholders. Compared to non-guided reading techniques such as *adhoc* reading, the PBR technique is more standardized and focused, because specific procedures of the review method can be established and each reviewer can focus on different aspects of the artifact. However, the PBR technique does not formalize a particular set of inspection methods used for every possible artifact but instead guides how the perspectives and methods can be created based on the software artifact [39].

PBR has been applied to various software documents. Because of the specific focus and the procedural guidelines, PBR techniques objective is to decrease the human influence on the inspection process, resulting in better cost-effectiveness ratios of the inspections in comparison with conventional reading methods.

2.3.1.4 Scenario based reading

Scenario based approaches [40], as the name suggests, uses scenarios to explain how to find the required information in a software artifact, and how this information should appear. This reading technique indicates the inspector a way to find the expected information in a software artifact. The scenario represents a limited and specific group of questions and includes detailed instructions on how the inspector should inspect the artifact.

In a scenario, the reviewer's attention is drawn only to a particular type of defects. To increase the overall team effectiveness, different scenarios are used, resulting in fewer overlapping defects detected[32].

2.3.2 Using tools to augment code review

As with many software development challenges [41], the need of making the process more accessible is a matter of great importance. Consequently, several code review tools have been developed to help developers with this process. Although code reviews tools are probably not going to entirely replace the formal code inspection process, using these review tools is not necessarily a downside [42].

Good tools can increase both the quantity and quality of reviews. To achieve this, they need to be flexible and lightweight, so team members do not view the tool as being part of the problem instead of part of the solution [42]. The following characteristics are expected in an effective code review support tool:

- Create code reviews easily.
- Require minimal bureaucratic effort.
- Keep participants notified of review changes and results.
- Quickly obtain review comments in context.
- Allow joined discussions.
- Facilitate formal review process established.
- Enable customizable metrics.
- Produce comprehensive reports of the code review results.

2.3.3 Participant selection

To maximize the potential of a code review, it is essential to choose the right code reviewers, as well as, implementing a good work culture [43].

In a recent study, different inspectors have proven to generate large variation in the effectiveness of software inspections. This happened even when they are using the same process of inspection on the same artifact [44]. In Rigby et al. study, [45] they suggested that experts and co-developers should review the artifacts because they already understand the context in which a change has been made. Checklists and reading techniques might force inspectors to focus during an inspection, but they will not turn a novice or incompetent inspector into an expert. Unsurprisingly,

without explicit knowledge of the project, reviewers cannot reasonably be expected to understand significant and complex artifacts they have never seen before [45]. Bosu's analysis [46] of the effect of experience in reviewing a file over five different projects, presented a substantial impact on the density of relevant comments (see Figure 2.5). Hence, if a reviewer had previously reviewed a file, his feedback would be twice more useful (65% -71%) than the first time reviewers (32% -37%).

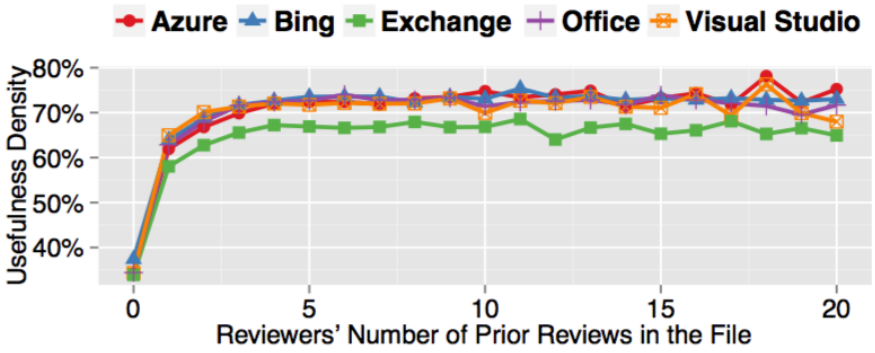


Figure 2.5: Previous experience reviewing the file vs. useful comment density [46]

Based on these results, it is possible to conclude that developers who already had a previous intervention on an artifact would provide more relevant feedback.

A principle based in the inspection literature is that two reviewers can detect an optimal number of defects. The number of defects detects by addition reviewers does not justify their cost [47]. However, knowing who are the best reviewers for a specific artifact is a frequent problem faced by authors of source code changes [48]. To approach this obstacle, authors of changes can use innovative algorithms, such as *cHRev* [49], *REVFINDER* [50], *xFinder* [51], to automatically suggest reviewers who are best suited to participate in a given review, based on their previous contributions proved in their prior reviews [46].

2.4 Grounded theory

Grounded theory research approach was used while collecting data about code review at *OutSystems*. It was originated by Barney G. Glaser and Anselm L. Strauss in 1967. This emerged as a way to face concurrent data collection and analysis for qualitative research. Grounded theory is an inductive, comparative, iterative and interactive methodology with systematic guidelines for gathering and analyze data to generate a theory [52].

In order to effectively use the *grounded theory*, some of Glaser and Strauss's [53][54] guidelines must be adopted. First, crucial coding practices lay the foundation of grounded theory research. Second, writing progressively more analytic memos instead of descriptive, advances grounded theory practice. Third, a crucial but often neglected grounded theory procedure, theoretical sampling, distinguishes grounded theory from other methods. Fourth, theoretical saturation is widely claimed but scarcely practiced [55].

2.4.1 Coding data

Coding is the first process in the data analysis. This methodology is used to analyze content and understand the problems in the noise found in the data. During the analysis of an interview, the researcher has to be attentive to expressions that highlight issues of importance or interest to the research used by the interviewee; these are then written and described briefly. If the same issue is cited again using similar words and is written again.

2.4.2 Memo writing

After collecting the data, then the next step is to write memos. Successive memos on the same category enable the researcher to gather more data to illuminate the category deeper into its analysis. By writing memos, the researcher also has the opportunity to learn about the data instead of just summing up material.

2.4.3 Theoretical sampling

By doing theoretical sampling we can maintain the study grounded. This method is used for sampling data for the development of a theoretical category. During theoretical sampling, grounded theorists' job is to fill out the properties of their specific categories.

2.4.4 Theoretical saturation

Theoretical saturation occurs when a theoretical category has been saturated. Researchers explain theoretical saturation as when gathering more data adds no further value on a theoretical category. On the other hand, most of the other qualitative researchers that do not use grounded theory speak of "saturation"of data as the same points repeatedly occurring in their data.

2.5 Human-computer interaction techniques

This section goes over several Human computer interaction (HCI) techniques used for evaluation the proposed solution. Note that this section describes the original techniques without any alterations.

2.5.1 Heuristic evaluation

Jakob Nielsen with the help of Rolf Molich in 1990 [56] have put together a set of usability heuristics. The objective of this method of evaluation is to easily identify issues related with the design of the User interface (UI). These were named heuristics because they are used as general rules-of-thumb and not as specific usability guidelines.

Here are the Jakob Nielsen's ten heuristics for UI design as published in his book [57], reproduced here verbatim:

Visibility of system status

"The system should always keep users informed about what is going on, through appropriate feedback within reasonable time."

Match between system and the real world

"The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order."

User control and freedom

"Users often choose system functions by mistake and will need a clearly marked 'emergency exit' to leave the unwanted state without having to go through an extended dialogue. Support undo and redo."

Consistency and standards

"Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions."

Error prevention

"Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action."

Recognition rather than recall

"Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate."

Flexibility and efficiency of use

"Accelerators — unseen by the novice user — may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions."

Aesthetic and minimalist design

"Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility."

Help users recognize, diagnose, and recover from errors

"Error messages should be expressed in plain language (no codes), precisely indicating the problem, and constructively suggest a solution."

Help and documentation

"Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out and not be too large."

2.5.1.1 Application

This method can be applied in several phases of the product, from its paper phase to the final product. However, it hold more value in the beginning and in the intermediate stages, raising many trivial problems, before testing with users and discovering the less trivial ones.

The output/deliverable is a simple list of problems identified. Each has a severity associated with it (for example, slight, serious, very serious), which allows the definition of implementation priorities.

Nielsen recommends that the analysis be done by three to five evaluators who are familiar with the concepts of usability, but people previously instructed on the evaluation criteria can also be evaluators [58].

2.5.1.2 Calculating the score

The evaluator can freely explore the system and then report the problems encountered, preferentially relating them to the heuristic criteria and classifying them according to severity—between 0 (not considered a usability problem) and 4 (a very serious problem that does not allow the completion of a task). In other words, each problem found in the interfaces is attributed the value of the severity of proposed by Nielson [58].

- 0: Not entirely viewed as usability problem
- 1: Only an aesthetic problem: Does not need to be fixed unless extra time is available in the project
- 2: Lesser usability problem: The priority of this problem should be low.
- 3: Greater usability problem: It is important to fix this, and it should be given high priority
- 4: Usability catastrophe: It is mandatory to repair it before the product is released

The formula used to calculate each heuristic's score is the following overall score is calculated by the following:

$$100 - \left[100 \times \sum SeverityValues_{PerHeuristic} \div \left(nProblems \times Max_{Severity}\right)\right] \tag{2.1}$$

2.5.2 System Usability Scale

The System usability scale (SUS) was developed in 1986 by John Brooke. Brooke's system consists of ten simple questions, and it was created as a "quick and dirty"scale for measuring usability. [59].

The usability questions are generally presented to the respondent after the completion of the test but before any debriefing or discussion. For more accurate answers it is better if the respondent gives a direct response to each item, rather than thinking about it for too long.

All questions should be checked. If a respondent feels that they cannot respond to a particular question, they should mark the center point of the scale. The center point of the scale is seen as a neutral value for the final score.

2.5.2.1 The questionnaire

SUS questionnaire contains ten items:

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

The items alternate between positive and negative. The respondent should be presented with this exact order of items to prevent response biases caused by respondents not having to think about each item [59]. The answer box is presented in Figure 2.6. The original questions were meant for industrial systems but these can be easily adapted to other fields.

2.5.2.2 Calculating the score

SUS return a single number representing the overall usability score of the system being studied. Important to note that the items' scores individually are meaningless [59].

To calculate the SUS final score, substract 1 from all the odd items contribution and subtract 5 from the even number contribution. Then, sum the result of the previous step and multiply that by 2,5. This yields the final score which will vary between 0 and 100 [59].

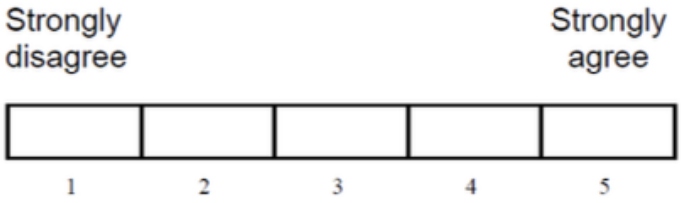


Figure 2.6: SUS answer format

2.5.2.3 Interpreting the SUS score

Although the SUS score is a value between 0 and 100, interpreting it as a percentage accurate. Bangor's study [60] showed that the average SUS score was of 70,14. Given an example of a system with 50% SUS score, as a percentage would be considered as an average system. However, it is located well above the average score of 70,14.

To avoid miss interpretations, it is recommended to normalize the score produced as a percentile ranking. So taking into consideration the 70,14 average score, normalizing the systems with a score of 50 indicates its usability percentage is 16,94%. Suggesting it is a below average system.

2.5.3 NASA Task Load Index

NASA Task Load Index (NASA-TLX) is a subjective and multidimentional assessment tool for rating perceived workload. Its can be applied in a variaty of complex socio-technical domains [61]. The term workload represents the cost of completing a task and there are many psychological definitions on how to measure it [62]. NASA-TLX measures workload by dividing it into six subclasses, known as scales: Mental, Physical, and Temporal Demands, Frustration, Effort and Performance [63]. The rating is a numeric value between 5 and 100, a low rating means a low workload and a high rating meaning a high workload.

The NASA Task Load Index (NASA-TLX) is a widely used, subjective, multidimensional assessment tool that rates perceived workload in order to assess a task, system, or team's effectiveness or other aspects of performance

2.5.3.1 The scales

Each scale should always be presented with a description and the participant should read each description with attention before rating it.

Mental Demand. How much mental and perceptual activity was required (e.g. thinking, deciding, calculating, remembering, looking, searching, etc)? Was the task easy or demanding, simple or complex, exacting or forgiving?

Physical Demand. How much physical activity was required (e.g. pushing, pulling, turning, controlling, activating, etc)? Was the task easy or demanding, slow or brisk, slack or strenuous, restful or laborious?

Temporal Demand. How much time pressure did you feel due to the rate of pace at which the tasks or task elements occurred? Was the pace slow and leisurely or rapid and frantic?

Performance. How successful do you think you were in accomplishing the goals of the task set by the experimenter (or yourself)? How satisfied were you with your performance in accomplishing these goals?

Effort. How hard did you have to work (mentally and physically) to accomplish your level of performance?

Frustration. How insecure, discouraged, irritated, stressed and annoyed versus secure, gratified, content, relaxed and complacent did you feel during the task?

2.5.3.2 The questionnaire

The NASA-TLX questionnaire is given to the participant after he completes a task. It is divided into two phases. In the first phase participant is presented with the six scales (with an accompanying description) and is asked to rate each scale within a 100-point range with a 5-point step [64]. An example is shown in Figure 2.7.

Figure 2.7: NASA-TLX answer format

The second phase asks the participant to weigh each scale's importance. With the six scales there are 15 possible pairwise comparisons, each of these are presented to the participant who chooses which scale he/she considers to be more relevant for the task. With this, each scale has a weight that ranges from 0 (not at all relevant) to 5 (more relevant that any other scale) [65].

To then calculate the final score, for each scale multiply its rating with its weight, divide that value by 15 and finally sum all the scales. The score will always be a value between 0 and 100 [65].

2.5.3.3 Raw-TLX

There is a version of NASA-TLX called Raw-TLX which removes the weighing process and the ratings are simply averaged. When using Raw-TLX, scales can be removed if they are considering irrelevant to the task. There are studies defending that Raw-TLX is better than the original [66], others defend that the original is better and others saying they are the same. With this, the choice comes down to personal preference [63].

Chapter 3

Related work

In this chapter, we describe the current code review methods including tool and processes used by some of the top software development companies. Moreover, we will talk about the expectations, outcomes and challenges of the modern code review.

3.1 Current methods

The methodology used for code review can vary even within a project. Therefore it is normal that different companies have different approaches. In this Section, we will describe few of them, including code review practices at *OutSystems*.

3.1.1 Microsoft

Microsoft [67] developed *CodeFlow*, a code review too used internally by code more than 50.000 developers.

3.1.1.1 CodeFlow

CodeFlow, described in more depth by Bacchelli and Bird [30], is code review tool similar to other popular review collaborative tools such as Gerrit [68], Phabricator [69], and ReviewBoard [70]. Figure 3.1, shows an example of a code review using CodeFlow. In it, several important features for code review are shown. These are: list of files to be reviewed(A), the participants and their status (B), a view of the current file with highlighted differences (C), all the comments given during the review (D) and different tabs for the individual iterations of the review (E).

3.1.1.2 Process

The steps an artifact has to go through in CodeFlow are rather straightforward. It begins with the author submitting the changes to be reviewed. The reviewers receive a notification via email and can start examining the change using the tool. Figure 3.1-F shows the overlayed windows that appears when the reviewer highlighting a portion

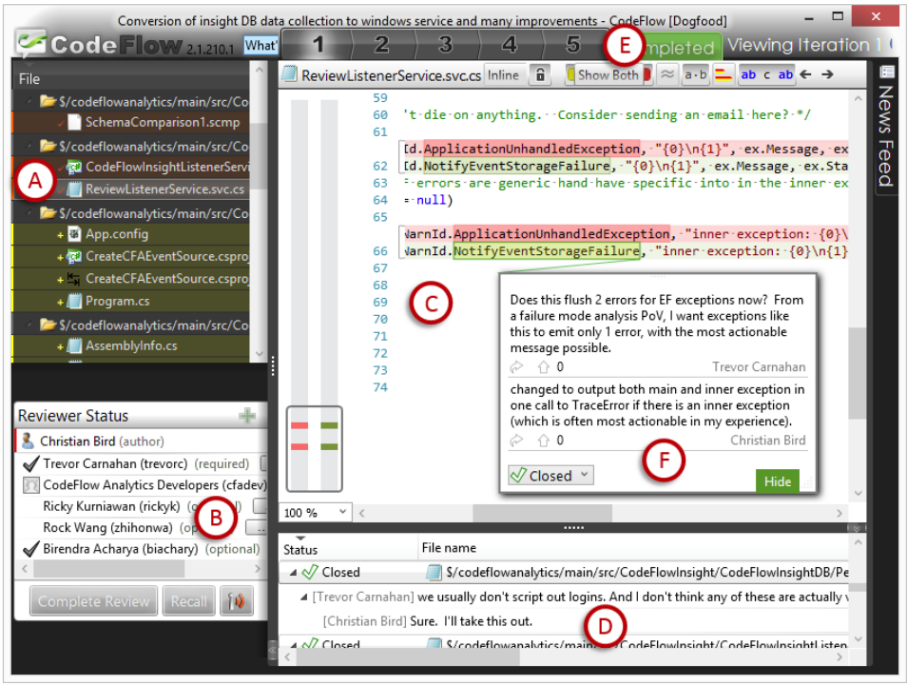


Figure 3.1: Example of Code Review using CodeFlow [46]

of the code to provide feedback. In line 66 we can see the feature described. Every time a user comments a new line, a new conversation is opened and participants can exchange their points f view. Every conversation has its status and it can be changed by the participants over the course of the review. The conversation default status is ‘Active’, but can be changed by anyone to:

- "Pending"
- "Resolved"
- "Won’t Fix"
- "Closed"

By not having a strict definition for each status and no enforced policies to resolve or close threads of discussion, teams can use them freely to track work status as needed. After the feedback has been given, the author may need to fix a defect, and submit the fixed changes to gather additional feedback. In CodeFlow, every time a change update is submitted for review it is considered an iteration and starts another review cycle. Before being ready to be processed to the main repository, the source code may need a few iterations.

In each iteration, the reviewers can continuously share their feedback using comments and this process it repeated until reviewers are happy with the quality of the code (sign off policies can differ according to the team). Finally, when the review is completed the author commits the changes into main repository [31].

3.1.2 Cisco

Cisco [71] uses Smart Bear Software's Code Collaborator tool [72] to assist the code review process within the company. The process and the tool have been studied by Cohen [73] and will be described further ahead.

3.1.2.1 Code Collaborator

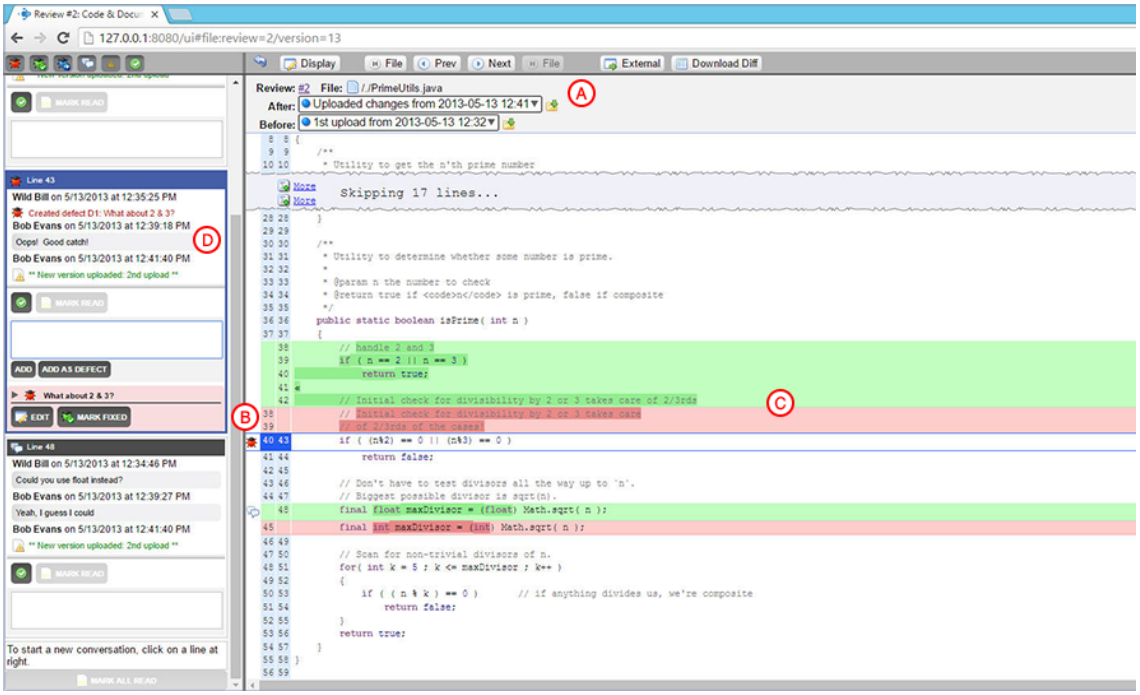


Figure 3.2: Example of Code Review using Collaborator [74]

3.1.2.2 Process

Cisco performs reviews their code before the changes are commited into the source code repository, the so called pre-commit. For this, authors invite reviewers. About half of these reviews only have one reviewer, while the remaining have more than one reviewers assigned [73]. Code Collaborator notifies the participants via e-mail. Code Collaborator presents a diff views of the previous/current version of the artifact to participants. Any participant can give their feedback clicking on a line of code and writing their comment. Defects are saved as comments and each one is associated with a line and file. When the author has fixed the defect, the can upload the new files to the same review. The tool then compares the new changes against the original and the reviewers can start a new review cycle. This iterative process can happen as many times as necessary for all defects to be fixed. Once all reviewers agree the review is complete and no defects are still open, the review is marked as completed and the author is then allowed to check the changes into the main source code.

3.1.3 Google

Google engineers have built their own Open source software (OSS) web-based code review tool called Gerrit [68]. It is a git specific implementation of the code review practices used internally at Google [45].

3.1.3.1 Gerrit

Gerrit is a lightweight framework intended to review every commit. It acts as a barrier between a developer's repository and the shared centralized repository. Its integration with email, allows authors to request a review (A), and reviewers to view side-by-side diffs and comment on them.

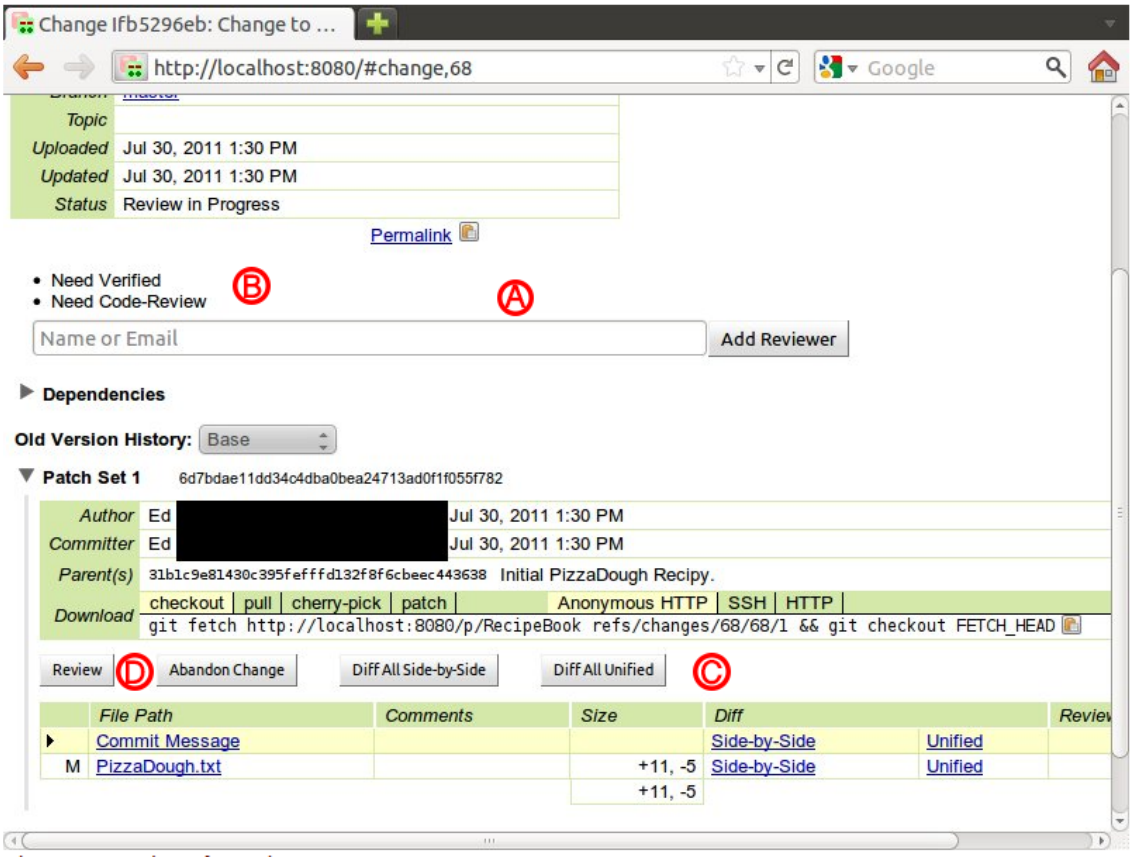


Figure 3.3: Gerrit Code Review Screen [68]

Figure 3.3 shows a Gerrit code review screen viewed by both author and reviewer. Here we can see both default requirement (B) demanded by Gerrit for a change to be accepted.

- **Need Verified.** Verifying is checking that the code actually compiles, unit tests pass etc. Verification is usually done by automatically by a server rather than a person.

- **Need Code-Review.** Code-Review is someone looking at the code, ensuring it meets the project guidelines, intent etc.

A reviewer can review the artifact within the Gerrit web interface as either a unified or side-by-side diff by selecting the appropriate option (C). In either of these views the reviewer can add inline comments by double clicking on the line that he wants to comment on.

Once a reviewer has looked over the changes he/she needs to complete reviewing the submission. The reviewer is able to do this by clicking the Review button (D) on the change screen where we started. This allows the reviewer to enter a Code Review label and message.

Once published these comments are accessible to all, allowing participants to discuss the changes.

3.1.3.2 Process

Every time a developer makes a local changes and then submit the changes for review. When the author of a change creates a new code review, the participants are notified by e-mail, with a link to the web-based review tool's page for that specific change. Notifications are sent via email every time reviewers submit their review comments. Reviewers can make their comments via the Gerrit web interface. For a change to be merged into the master branch, it has to be approved and verified by a senior developer. Google has a pre-commit policy that has additional change approval steps:

1. **Verified.** Before a review begins, someone must verify that the change merges with the current master branch and does not break anything. In many cases, this step is done automatically.

2. **Approved.** Anyone can review a change, however only someone with appropriate privileges and expertise (a senior developer, typically) can approve the change.

3. **Submitted/Merged.** Only when the change has been approved, the code is merged into Google's master branch so it can be accessed by other developers.

Google has tools for automatically suggesting reviewers for a given change, by looking at the ownership and authorship of the code being modified, the history of recent reviewers, and the number of code reviews for each potential reviewer. At least one of the owners of each branch which a change affects must review and approve that change. But apart from that, the author is free to choose reviewers as they see fit [75].

3.1.4 OutSystems

At *OutSystems* code review is separated in two different approaches. Unlike other companies, besides the traditional code review on text based programming language, *OutSystems* also perform code review on artifacts developed using their visual environment, *Service Studio*.

3.1.4.1 Code review of textual artifact

Contrarily to the examples in, Subsections 3.1.1, 3.1.2 and 3.1.3, *OutSystems*'s code review begins when the changes are committed, in other words, post-commit. Teams use Upsource [76] tool to assist code review. Invited participants (reviewers and

optionally watchers) (A) are notified by the tool via email. In a typical code review there are one or two reviewers involved; a team member that has context on the change. In the case of higher complexity, a member with more knowledge in that specific context is invited to join the review. To avoid unnecessary workload, the invitation is only sent when reaching the final iterations. The next step on a code review is the actual examination of the code. The Upsource's web based interface (Figure 3.4) allows to view all files involved in the review (E) and provide feedback by placing comments over any selection of the source code. Comments are displayed between the selected lines (C). Issues, if existent, are discussed by way of exchanging comments with the reviewers. Upsource provides unified and side-by-side diff views (B). It also provides the possibility to switch between the diffs of the versions included in the review (D). The typical code review work flow is: resolving the issues, commit the fixes, and add the new revision to the existing review. If there are no revisions left, as soon as reviewers approve the changes, the review is closed.

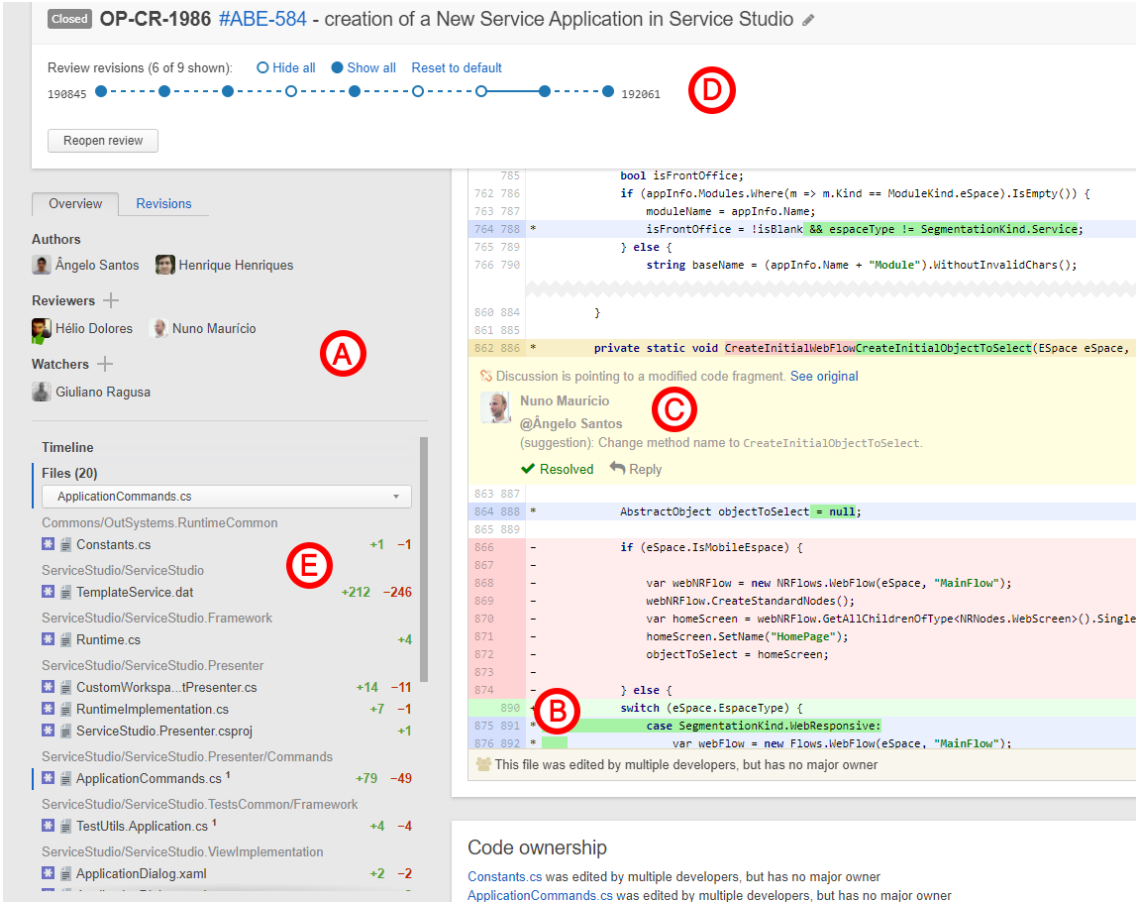


Figure 3.4: Example of Code Review using Upsource

3.1.4.2 Code review of visual artifacts

As the visual programming field gains momentum, the need for code reviewing it also rises. However, considering it is a somewhat recent way of development, the techniques to perform its code review are not as systematized as the traditional code

review.

This issue is also present at OutSystems, as different teams have different ways of approaching code review on visual artifacts. The overall process is mainly *ad-hoc*. On the one hand, there are the teams which follow a check-list of best practices to make sure the company's standards are followed. On the other hand, the ones that perform code review over the shoulder at the authors' desk, giving instantaneous feedback.

The main problem which the developers at *OutSystems* are faced with is to understand the context in which a change was made (i.e. the change was made by X and is related to the code review issue Y). Developing in *OutSystems'* IDE the developer publishes in the server. The server is the repository of all the versions. Figure 3.5 is a representation of the single branch type of repository used by Service Studio. The different icons represent different developers. There is no concept of commit. When a developer publishes, other developers are publishing at the same time. This results in an accumulation of unrelated changes when a reviewer tries to review the change, making the code review process harder.

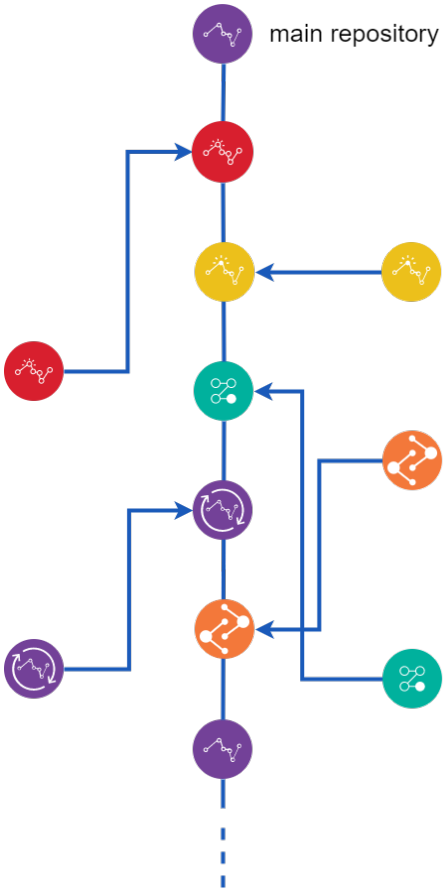


Figure 3.5: Single branch repository used by Service Studio

3.2 Summary

Code review is heavily present in some of the big software development companies. However, the type of code review can vary from company to company.

By adopting post-commit review OutSystems allows their developers to work and commit changes to the repository continuously while other team members can see the code changes and can alter their work accordingly. However, this type of review will increase chances of poor code being inserted into the main repository, thus affecting the entire team's work.

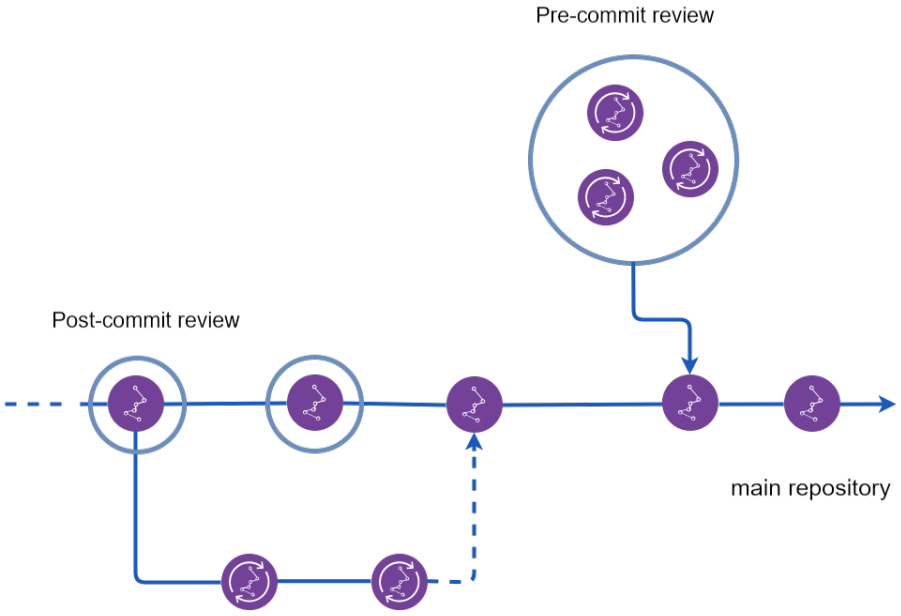


Figure 3.6: Pre-commit vs Post-commit review

The companies mentioned in Subsections 3.1.3, 3.1.1 and 3.1.2 by performing pre-commit review, ensure that developers in their team will not be affected by bugs that may be found during a review. At the same time, the company's coding quality standards are met before the work is committed to the main repository and reviews are not postponed or omitted. On the other hand this anticipated process can decrease productivity of each developer, since further work on the submitted code is impossible until a successful review.

In Figure 3.6 it is possible to understand the difference mentioned in the above paragraphs.

In relation to the tools, it is common practice of big companies to use them to ease the code review process. This allows developers to create reviews and explain the changes while they are fresh in their heads. Without theses tools, they either would need to bother someone to review the code or switch context to explain the modification when they have already moved on to the another task. This is also particularly useful with remote teams where this process can not happen face-to-face. Even for a co-located team, having available a review tool is beneficial because reviews can be done at a time that is convenient for the reviewer while maintaining . Generally, these tools are able to store code review metrics and generate statistics that can be useful for company's studies.

Every company has different policies for selecting participants. For instance, Microsoft chooses participants automatically with the help of parametric algorithm [77] [49]. Unfortunately, for the rest of the companies mentioned in section 3.1, no specific information was found in literature.

Finally, one thing all four companies have in common is the belief in peer code review importance. All of them use this process as a method to detecting and removing bugs, increasing overall understanding, fixing design problems, and learning from one another.

Chapter 4

Requirements elicitation

In this chapter, we will explain each step of procedure followed to gather the requirements necessary to propose a new solution for reviewing VPLs.

4.1 Procedure

The elicitation procedure followed the iterative process shown in Figure 4.1. The first step was interviewing developers and understanding where and why they were having trouble when inspecting visual artifacts. The next step was bench-marking the leading software development companies and the strategies used to augment the quantity and the quality of their code. Finally, creating a mock up interface to be able to test and make sure our proposal really fitted the existent needs. The requirement elicitation process only ended upon reaching satisfactory results.

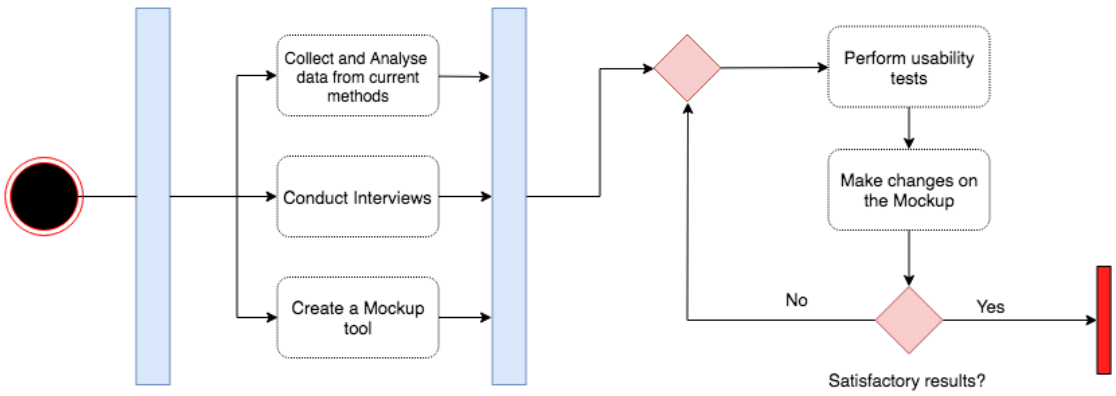


Figure 4.1: Requirement elicitation work-flow

4.1.1 Interviews

During the course of this work, a group of 10 developers from diversified teams and different seniority levels within OutSystems were interviewed. All the interviews conducted were inspired in a simplified version of grounded theory approach [54] and concluded upon the gathered data reaching a saturation level.

The research focused on in-depth understanding of the code review process on the visual environment within the company. The aim of the study was to interpret, analyze, and understand OutSystems' current methods and problems on code reviewing visual artifacts from a developer's perspective.

The interviews did not follow a predetermined script, rather the interview followed the grounded theory method [54]. The goal was to find the root of the problem by getting the experts to tell a story.

The main topics addressed during the interviews were related to the code review process. Understanding if the developers performed code review, how it was done, what kind of mechanism were available. The full script can be found in Annex I.

Even though some of these topics are binary questions (which goes against the grounded theory philosophy) by letting the interview unroll freely, we were able to expand the topics and collect insights about them.

There are a number of important assurances of quality in keeping with grounded theory procedures and general principles of qualitative research. The following points describe what was crucial for this study to achieve quality.

During data collection:

1. All interviews were digitally recorded, carefully transcribed in detail and the transcripts checked against the recordings.

2. We analyzed the interview transcripts as soon as possible after each round of interviews. Under these circumstances, we are able to decide what data we still needed and who we wanted to interview more, allowing the theoretical sampling to occur.

3. Writing memos after each interview allowed to capture the main ideas and make comparisons between participants.

4. Having the opportunity to contact participants after interviews to clarify concepts and to interview some participants more than once, contributed to the refinement of theoretical concepts, thus forming part of theoretical sampling.

Through our analysis, the following insights were extracted from the interviews:

- Code review process not well defined or sometimes non-existent. (referred by 10 of 10 interviewee)

- No dedicated tools to promote and facilitate code review. (10/10)

- Platform server centralized repository makes it difficult to understand the context of a change.(10/10)

- No possibility of branching to allow coordinating all the team separately and isolated. (9/10)
- Developers do not have the possibility of performing pre-commit reviews, given OutSystems publish and commit system. (8/10)
- Changes sometimes considered too big. (4/10)

4.1.1.1 Discussion

Grounded theory methods were useful in providing deep insights and understanding of the current method of code review on the visual development environment. With this analysis, it became clear that OutSystems lacks resources and mechanism to conduct code review on visual artifacts. From the interviewees perspective, having such support would drastically change their approach toward code review. Also, the fact of not having a well-defined process within the company make this quality assurance technique very dependent on the participant experience.

Moreover, the current structure of the platform and the post-commit review policy increasingly hinders the process, making it impossible to track changes by an author. Reviewers are forced to review two different versions containing changes done by different developers. Reviewers do not have a way of understanding the circumstances of the change.

Information such as:

- Who changed it?
- What changed?
- Where it changed?
- Why it changed?
- When it changed?

Are all impossible to access with the current code review system which makes reviewers job harder.

4.1.2 Benchmark

Even tough no code reviewing methods or tools were found for VPLs, benchmarking the leading software development companies and the strategies used to augment the quantity and the quality of textual code reviewed was important to complement the research.

The process of benchmarking was based on Camp's procedure [78] and involved three distinct steps:

Identifying the companies/industries to benchmark, where code review is also a matter of importance. These areas of interest where chosen accordingly with Section 3.1, where a previous study of the current methods was conducted.

The second step that followed was collecting data from these identified companies and from the code review tool used by them.

Finally, when the data has been collected, it was important to study the gaps between the actual method of code reviewing visual languages at OutSystems and the ones used by the companies benchmarked.

In Figure 4.2, it is possible to observe the results extracted upon conclusion of the benchmark.

	Google	cisco	Microsoft	outsystems
Tool based	✓	✓	✓	✗
Iterations	✓	✓	✓	✗
Adding Participants	✓	✓	✓	✗
Chat	✗	✓	✓	✗
Diff-viewer	✓	✓	✓	✓
Review Status	✓	✓	✓	✗
Participants Recall	✗	✓	✓	✗
Files List	✓	✓	✓	✗
Web based	✓	✓	✗	✗

Figure 4.2: Software industry benchmark

4.1.2.1 Conclusion

Although this comparison showed that different companies have different methods and technology available, it also made clear that a large gap between visual and textual code reviewing resources existed. On the other hand, with this benchmark it is now possible to know what are the features desired in a code review tool, independently of the type of artifact. Even though some of them are only applicable to text, they can be adapted and used to facilitate the visual code review context, homogenizing the standard of performance for code review tools in the software industry.

4.1.3 Design Goals

In order to maximize the quality of the output generated from our solution, besides the technical requirements, it was also crucial to apply design goals. We consider the output to be of high quality if it satisfies the following four requirements:

Timely Feedback should be given to the author as soon as possible, in order to a have efficient code review process and provide faster feedback cycles for the author. Timely comments and answers promote real discussions between participants.

Relevant Comment should be relevant for the topic. Answers to comments should also be related to the topic's previous comments.

Correct Presenting accurate information is important to guarantee the efficiency of our system, as well as, the trust of users.

Informative It is important to, along with the identified issues, also provide solutions to help authors learn from their mistakes. This also means that we favor substantial comments with interesting information (e.g. "I would use List here instead of Array based on your implementation strategy") instead of superficially correct comments (e.g. "You misspelled a word in this comment.").

When designing our interface we tried to maximize these requirements by following well-established heuristics from Nielsen presented in Subsection 2.5.1 and by defining design strategies.

These strategies are:

- **Reduce overhead** When possible, contributing to the system should be as easy as possible. The interface should also attempt to minimize the amount of extraneous information and interface elements presented to the user.
- **Contribution importance** Users should have knowledge of the results of their contribution to quality software.
- **Present relevant information** The interface should, as much as possible, attempt to identify and present relevant and high quality content to the user. This helps the user learn from that content, and feeds back into the system in the form of more informed contributions.

Merging all this information together gave us enough background to build a model to acquire feedback from possible users.

4.2 Mock-up

Before starting implementing the tool, we wanted to make sure our proposal fitted the existent needs. A high fidelity mock-up interface was designed using Sketch software [79].

The mock-up was inspired both on features extracted from the current methods and existent code review tools. To complement and diversify our source of inspiration, we took advantage of the insights derived from the interviews conducted. This allowed us to have a more in-depth understating of the actual problem code reviewers are facing when VPLs are concerned.

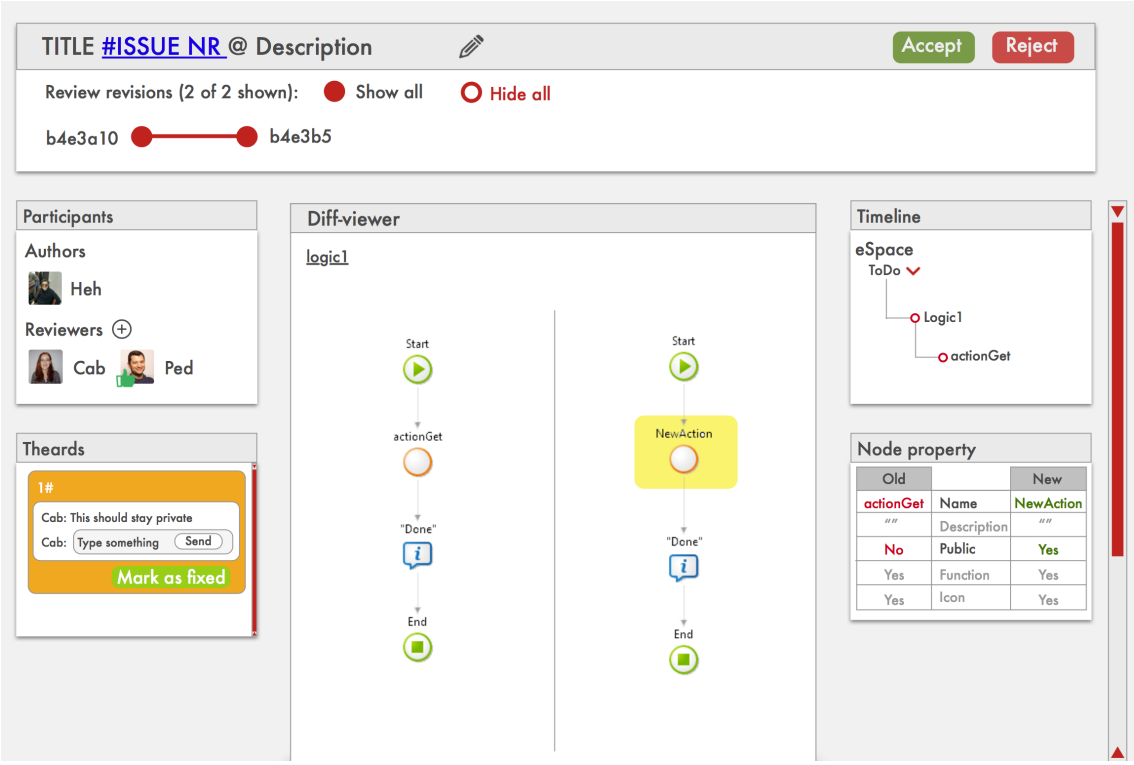


Figure 4.3: Mock up interface

This interface allowed users to test the interface and make relevant comments on the features before expending valuable time developing the actual product. Thus, minimizing time wasted and maximizing the overall efficiency of the tool. Testers' insights will be discussed further in this chapter.

4.2.1 Usability experiments

For the purpose of testing the mock-up interface, one-on-one experiments were performed with both experienced and inexperienced users. Once again, our goal was to extract as many insights as possible, prior to the development.

4.2.1.1 Experiment protocol

The population for this experiment was composed of 10 software developer and half of those had previous experience with code review.

We recreated a low complexity scenario so testers could focus more on the actual tool and process and not feel pressured to understand the artifact.

The experiment was set up on a computer screen (See figure 4.4). In the same windows, the testers would be able to see the set of heuristics defined by Nielsen [58] for easy recall.

Testers were asked, and upon touching the elements, new web screen prints were presented. At the same time, testers would report the problems encountered, relating them to one of the ten heuristic criteria and classifying each problem with a severity score ranging from 0 to 4.

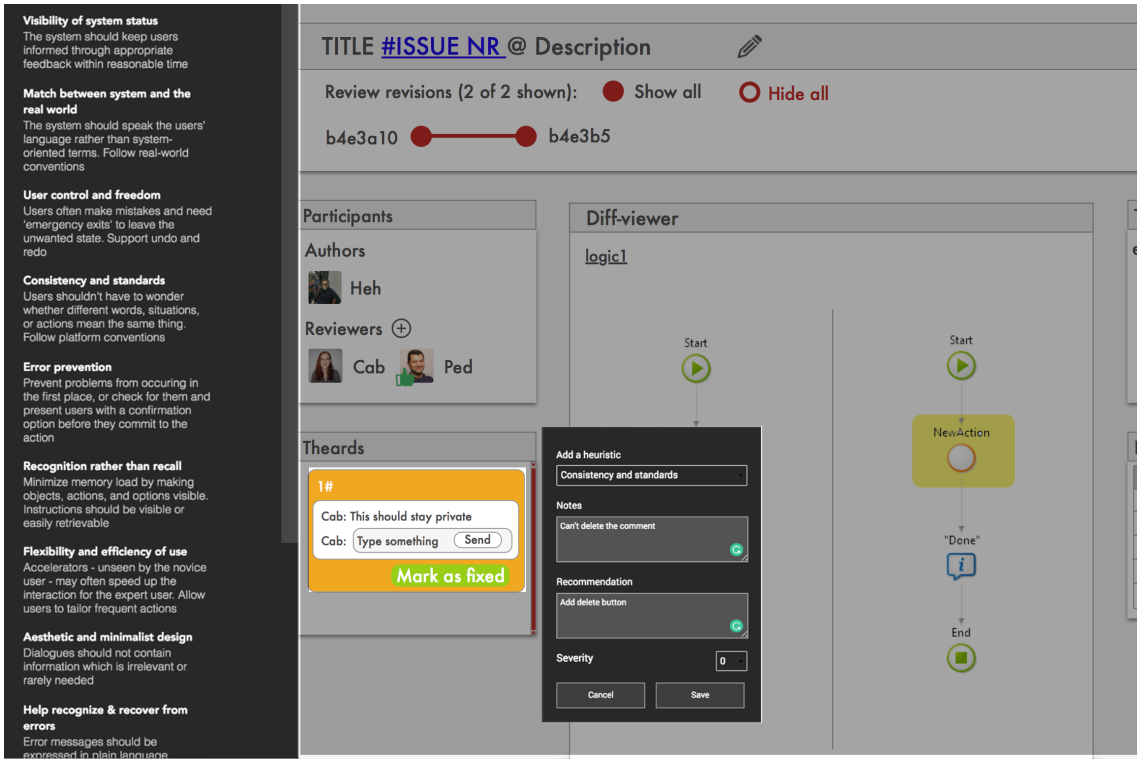
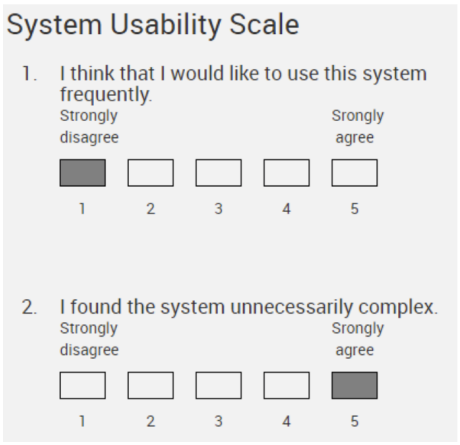


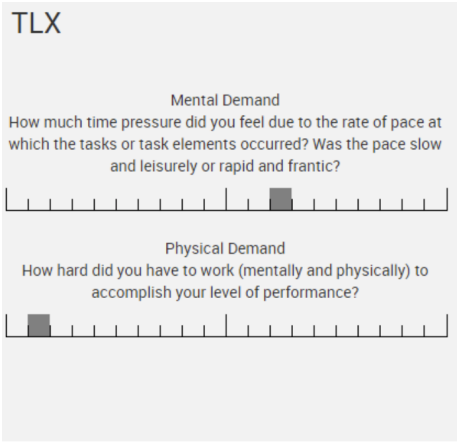
Figure 4.4: Example of Usability Experiment screen.

After the tester had gone through the whole tool, he would be asked to answer the SUS and TLX questionnaire shown in Figure 4.5.

At the end of the experiment, there was a small discussion where the user would talk about the issues they had with the process and the tool. For each issue, together with the tester, we tried to understand what caused the problem and what could be done to correct the issue in accordance with the users needs.



(a) SUS



(b) TLX raw

Figure 4.5: Example of questionnaire

4.2.1.2 Experiment analysis

The results of the analysis is a simple list of problems identified, each having a heuristic and a severity score (ranging from 0 to 4) associated. These last will be later used to define the priority of the changes and which where the weaknesses of the user interface.

The list of issues was compiled and for each a possible solution. Most notably were:

- **Selection.** (referred by 10 of 10 interviewees)

  Issue: All the users had difficulties getting feedback from the tool since most actions required double click.

  Possible solution: Changing to single click would resolve this matter.

- **Grouping.** (7/10)

  Issue: Most of the users would like to comment on portions of the code.

  Possible solution: Allowing to associate several node elements to a comment would be useful.

- **Process flow.** (6/10)

  Issue: Users felt that author should not be able close the review.

  Possible solution: Reviewers should be the ones setting the status of the review.

- **Association.** (6/10)

  Issue: A common issue encountered was not being able to understand the connection between the comment and the location in the code.

  Possible solution: By associating every feedback directly with a node element in the diff no more confusion existed.

- **Labeling.** (5/10)

  Issue: Half of the users thought error/warning label on the comment expressed confusion.

  Possible solution: Changing to important/comment would be more reliable.

- **Comment removal.** (2/10)

  Issue: Some users noticed the impossibility of deleting the given feedback.

  Possible solution: Adding a removed button would allow deleting comments created by mistake.

To complement our analysis, we tried to better understand the weakest areas of our UI using the heuristic criteria and the severity classification chosen by the testers for every issue identified. Using the formula specified in 2.5.1.2 we calculated the overall score of each heuristic and illustrated the results in a radar graph (Figure 4.6).

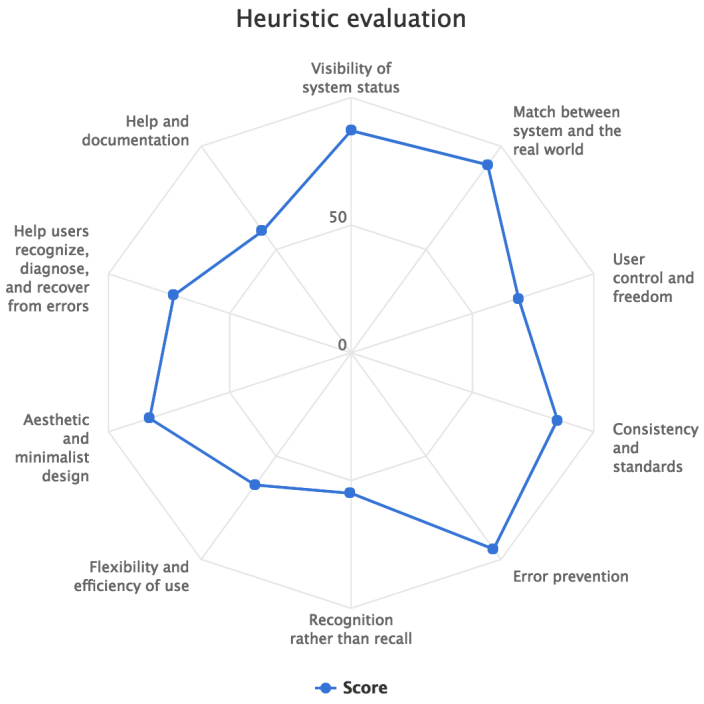


Figure 4.6: Heuristic issues distribution.

Representing this data visually helped us to quickly identify problem areas and which heuristic needs attention without looking through all heuristic question sets.

This graphs are useful because they give a recognizable shape based on the score. The more circular the radar, the more balanced the score; the spikier the radar, the more variation in the score. The size of the radar plot on the axes indicates the score percentage itself, longer axes showing better areas.

In our case, we were able to conclude that the documentation and elements recognition were our weaknesses, whereas the error prevention and the match between the real world were the strengths. For a final version, we knew fixing this problems was mandatory, so we decided to add an help menu, focused on explaining the steps for the users' task. Another important change would be addition of visual cues that would help the memory retrieval, such as adding flags next to the portion of code commented, so participants do not have to recall which comment was related to which part of the code.

Chapter 5

Proposed solution

In this chapter, we present the tool and the process proposed. We will begin by explaining the architecture of the solution, the types of input artifacts and code pre-processing. Further, we demonstrate its UI and explain all its features in greater detail. Finally, the code review process is explained.

5.1 Tool

We developed a web-based tool to assist the code review by providing a wide range of mechanisms previously not available to a VPL. The tool's features were inspired in both already existent software for textual code review and problems identified in a visual programming context at OutSystems, such as: not being able to comment on a specific change, not knowing the context or the order of the changes, etc.

5.1.1 Architecture

Figure 5.1 represents the architecture of our solution. It is composed of several components that work together to make the entire tool function accordingly.

The architecture is split to allow the logic to be refactored or replaced without impacting the UI.

The solution's architecture encapsulates two main components, making them work as one unique body from an outsider's perspective. It also includes converters that transform the visual languages artifact into a generic structure. All the components will be explained in detail further ahead.

5.1.1.1 Generic structure

One of the goals of the dissertation was to offer a generic code review tool. Although OutSystems language was used as a base throughout the development, the architecture of the proposed solution was created in such a way that different sources of input could be provided without compatibility issues. Hence, instead of directly consuming

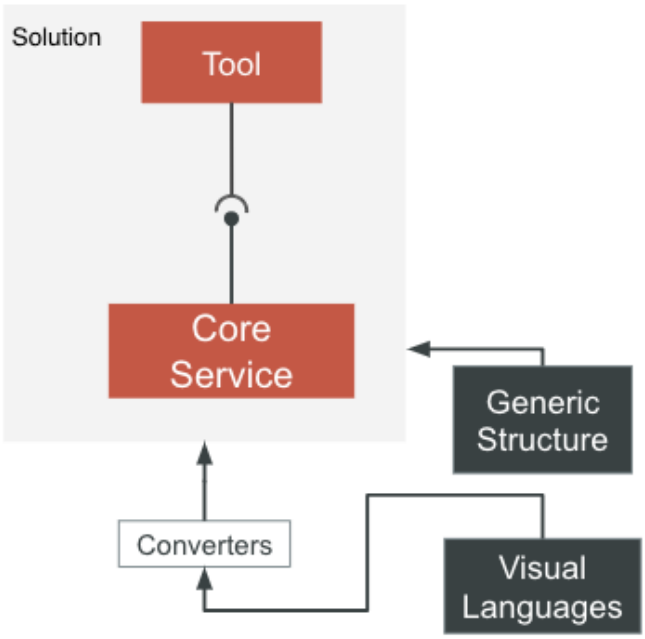


Figure 5.1: Solution's architecture.

the artifact imported from OutSystems language or any other visual language, we opted to create a generic data structure.

When we thought of VPLs, we realized most of the artifacts could be translated in a node model structure with links connecting all the elements. Therefore, to be able to manipulate and access the individual in formations of the visual elements, the structure should contain the relevant data of each element, including the in-going and out-going links.

Figure 5.2 shows a simple example of an artifact represented using the generic structure we defined. Its data includes an array of two artifacts, representing the previous and current versions; inside each object, their name, ID, icon, version, list of nodes. All the nodes are also represented using objects. Inside the node object, it has its own information, as well as, a list of IDs to the outgoing nodes.

Some of this fields are not mandatory and, if omitted the tool automatically adjusts itself. It is the case of the position coordinates, if not included the nodes will have centered relative position instead of absolute. The same thing for the icons, if the path is not included or valid, default icon is shown.

This step backward enabled us to build a solution capable of accepting not only OutSystems' code but also other visual languages. In fact, as said before most VPLs artifacts elements can be seen as nodes and can be converted into this structure, unless the elements do not have a unique identifier which is essential for the relationship between the nodes.

```
[
    {
        "FileName": "Artifact_XPTO",
        "versionId": "1.4",
        "icon": "~/img/xpto.png"
        "nodesList": [
            {
              "id": "xpto",
              "icon": "~/img/xptonode.png",
              "nodeType": "If",
              "XPos": 100,
              "YPos": 100,
              "properties": {
                  "label": "hasClient",
                  "condition": "getClient()!=null"
                },
              "outNodes": ["otpx", "yzxq"]
            }
        ]
    },
    {
        "FileName": "Artifact_XPTO",
        "versionId": "1.2",
        "icon": "~/img/xpto.png"
        "nodesList": [
        ]
    }
 ]
```

Figure 5.2: Tool readable JSON data structure.

Knowing that other VPLs could be supported if homogeneously converted into the tool readable structure was essential for the development of the tool because it allowed us to build all the logic on top of something static. By not having to worry about each language specificity, we were able to quickly develop a solution which we knew would be capable of supporting other VPLs.

5.1.1.2 Converters

The converter units present in our architecture are modules that have to be created to convert specific programming language code into the generic structure shown in Figure 5.2. This conversion enables artifacts to be later processed without any compatibility issues.

For the purpose of this dissertation and to be able to properly test the tool, we had to create the first converters. Since this work had OutSystems as a case study we developed a converter for their visual programming language.

However, to prove that the solution could be used for other visual languages, we decided to add another language besides Outsystems. Therefore, a second converter was created, and this time it received a UML Class Diagrams as inputs.

For both languages, the visual artifacts had to be iterated using their textual representation. Though, differently structured, the two languages had their artifacts textual representation made using XML language. For OutSystems, the file was associated with the structure they call "Action"and the node elements were their "Nodes". In

the case of UML, the file was the "Diagram"and the nodes the "Classes". Defining this notation, enable to associate elements of the VPLs to JSON structure elements. Finally, the artifacts were ready to be converted into the desired format, to be used by the core service later on.

Figure 5.3 shows the final result after the whole process had taken place. The UML class diagrams were converted into the generic structure. Then, the structure was consumed by the core services, and the differences were computed and highlighted.

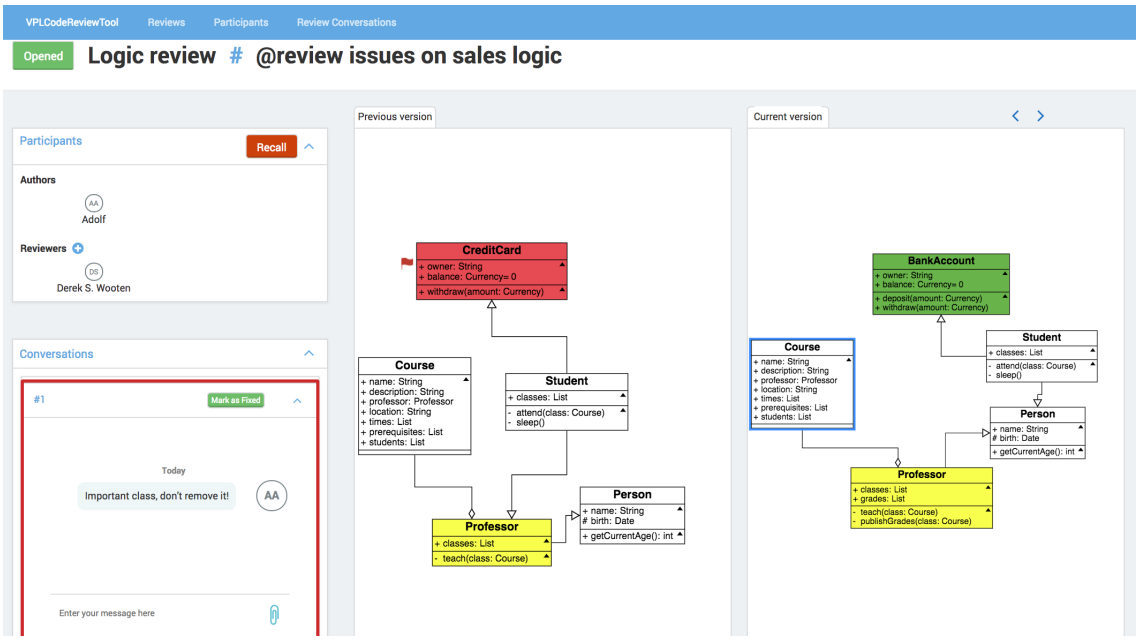


Figure 5.3: UML class diagram code review

To be able to support other visual languages, converters have to be created and added to the solution. A converter can be a simple program that receives two binary files, each being different versions of an artifact. These input files have to be iterated and parsed into the JSON structure file. This result must be the program's output.

All the new converters can easily be added as extensions through Integration Studio (See figure2.1.1.4). In figure 5.4, it is possible to observe the existent converters and how they are parameterized. The input parameters must be the two binary files, and the output the JSON conversion. Using this method it is possible to integrate new VPLs with the existing system without affecting any of the previous work done.

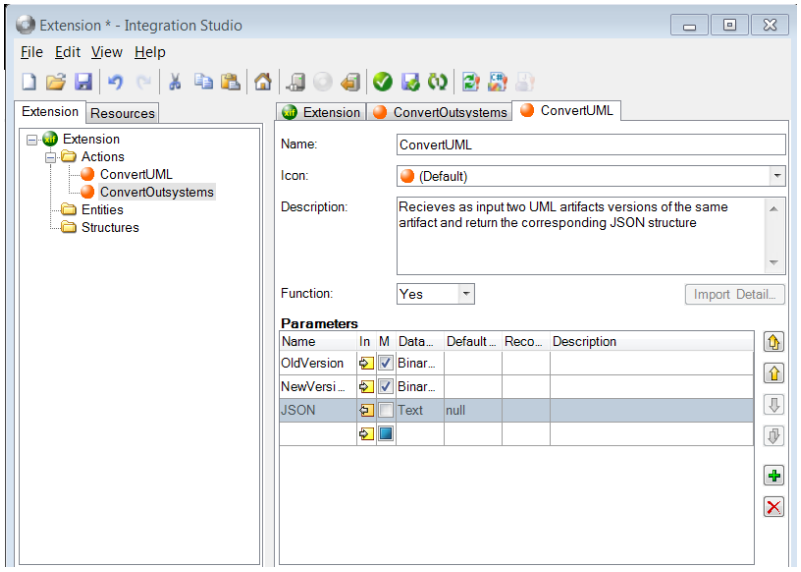


Figure 5.4: Integration Studio interface

Once added, the new VPLs will be available for usage and the author can create code reviews for an artifact of that new language. Figure 5.5 show the panel to create a new code review with the current VPLs available.

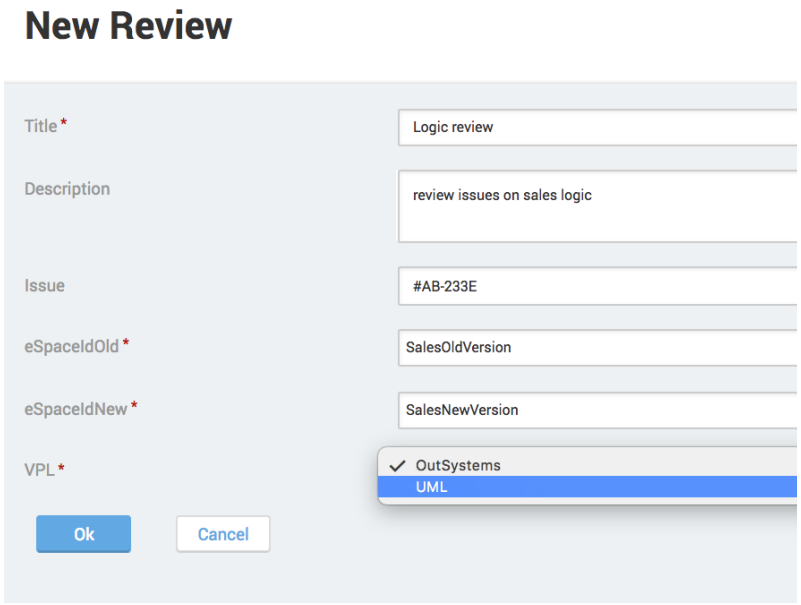


Figure 5.5: Existing converters

5.1.1.3 Core service

Core service is the entity that represents the backend of the tool. It is responsible for processing any input artifact as a generic structure and storing it into database elements. This pre-processing enables to have all the visual elements of the artifact saved individually. Algorithm 1 represents the pseudo code of the JSON structure parser. The function will iterate through all data elements included in the inputted

JSON. Starting from the highest level, the artifact, until the lowest level, the outgoing links of each node.

Algorithm 1 Pseudo code - Store JSON structure in database

```
if correctJSONFormat(json) then
  for all artifact in json do
    createNewArtifact(artifact)
    for all node in nodes do
      createNewNode(node, artifact)
      for all property in properties do
        createNewProperty(property, node)
      end for
      for all outnode in outNodes do
        createLink(node, outnode)
      end for
    end for
  end for
end if
```

This step is important because it allows the integration of all the tool features. These are:

- Relate single or multiple visual elements to a comment.
- Show detailed proprieties of individual elements.
- Iterate through changes in the chronological order of the modifications.
- Present different color highlight depending on the modified status.

Once this information is stored, the next step in this service is to compute all the differences between the two versions of the artifact. An iterative process is run simultaneously on both versions and updates each element status depending on their presence or absence in both versions.

Finally, all the data is ready to be presented side-by-side in a visual manner, along with all the features designed to help participants conduct code reviews.

5.1.2 User Interface

The user interface is the only component end users have access to. With its straightforward design and intuitive features, participants can profit from all the resources of a code review tool while focusing their attention on detecting defect and consequently increasing code quality.

Figure 5.6 shows the tool's interface, while reviewing a visual artifact.

The tool's features include:

- **Diff-viewer** a side-by-side view (A) of two different versions of the same artifact;
- **Changes highlight** different color highlighting on changes for quickly understanding what changed (G). Red for deleted elements, yellow for modified and green for added;

- **Threads of discussion** participants can select a single or multiple nodes and create new discussion threads (B);
- **Issue Tracking Software integration** JIRA Atlassian integration for more extended context on change (C);
- **Properties** all changes have a properties table (D) to more in-depth context of a specific change;
- **Files list** Tree listing of all files (H) related with the review;
- **Participants list** Ability of adding participants as needed (E);
- **Notifications** Automatic notifications to every participant whenever a review is updated;
- **Data** stores data of each review and provides metrics such as number of defects found, average time per review, etc.

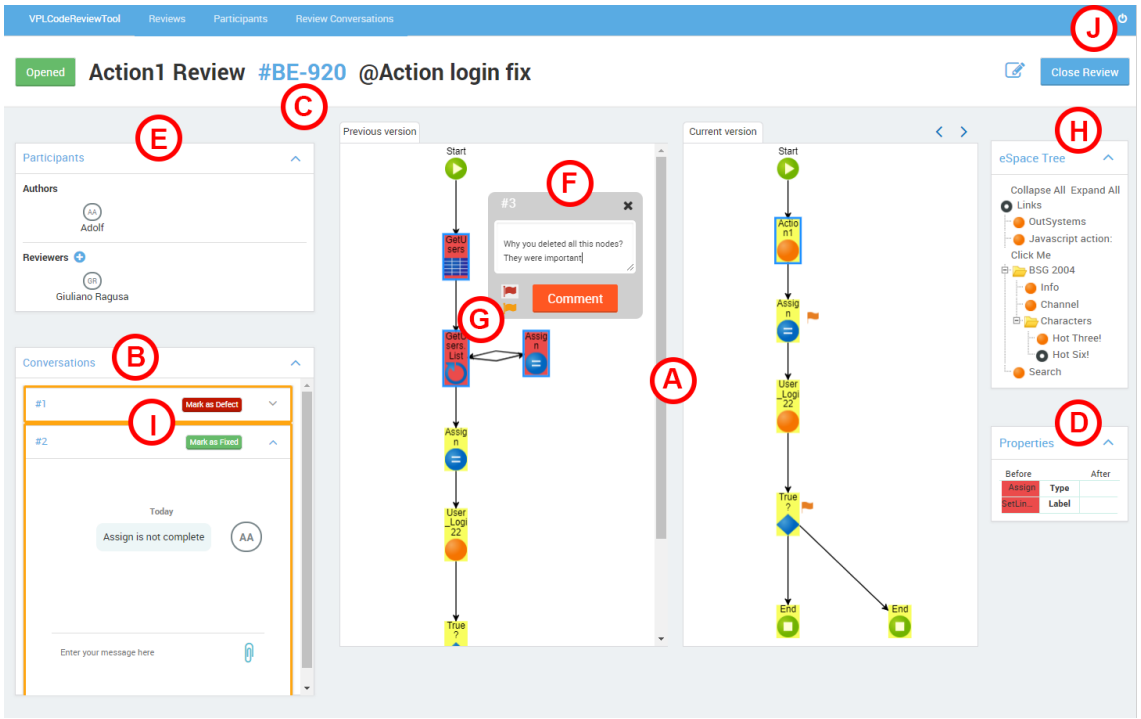


Figure 5.6: Tool's user interface.

The tool presents a side-by-side view with all the differences highlighted. The reviewer will be able to open threads of conversations by simply right clicking on any node and all threads opened will be listed on the side. The participant can choose the importance of his comment by switching the thread's flag from warning to critical. By clicking any highlighted change, the reviewer will be able to get all the information on that specific change, giving them a more accurate context. If needed, the reviewer can click in the issue hyperlink which will redirect him directly to the company's issue tracking platform. Reviews are only closed when they need no more iterations and

all critical threads of conversations are marked as fixed. After reviews are closed, reviewers can be more confident in the quality of their code even though it does not mean the artifact is invulnerable, since some bugs may have gone unnoticed.

Improving the changes' traceability was one of the main goals of the tool. In fact, this is important because the reviewer benefits from quickly understanding the circumstances of the change. With our tool, reviewers will instantly know which author made the change, on what purpose, what was the chronological order of the changes, and where and what has changed.

5.2 Process

The code review process was created to provide guidance to the tool's potential users. The process includes a work-flow where users can learn each role and their functions, explanation on how reviewers are suggested and how reading techniques should be addressed.

5.2.1 Work-flow

The swimlanes diagram in 4.1 represents the overview work-flow of this process, and for each action its corresponding agent.

The whole process starts when a developer has made a change on the code. As an author of the changes their first step after creating a new review is the participant selection.

As reviewers are chosen by the author, they are notified and they can begin the review with the help of the tool. One of the goals of our solution is to maintain the interaction between the participants and the tool. This will be done via email notifications. Every time a review is created, a participant finishes his review or a review is closed. It is important to keep notifying all participants of the review's updates, as users are prone to unintentionally forget them.

Participants will also be able to iterate through all the changes in a chronological order. This order is the key for the reviewer to step in the mind of the author, follow his flow and quickly understand the context of the changes. This can help reviewers to know what to look for and, more importantly, how to scrutinize the visual artifact, reducing the overhead.

When finding defects in the code it is important that reviewers do store feedback somewhere, otherwise it becomes useless for the author. So the process will also guide the reviewer to provide the feedback and enable the exchange of points of view between the reviewer and the author in shape of a conversation. Each conversation is directly associated with a single or multiple parts of the code and a priority flag. When the reviewer initially creates the conversation by commenting the code, he can choose between warning or critical priority. This flag is added to the diff canvas, right next to the corresponding area, so other participants are able to relate parts of the code with feedback already given.

After giving all the feedback and the reviewer has chosen to finish his review, the author is notified. He can then start fixing the defects or comment on the existing conversation if he does not agree with the feedback.

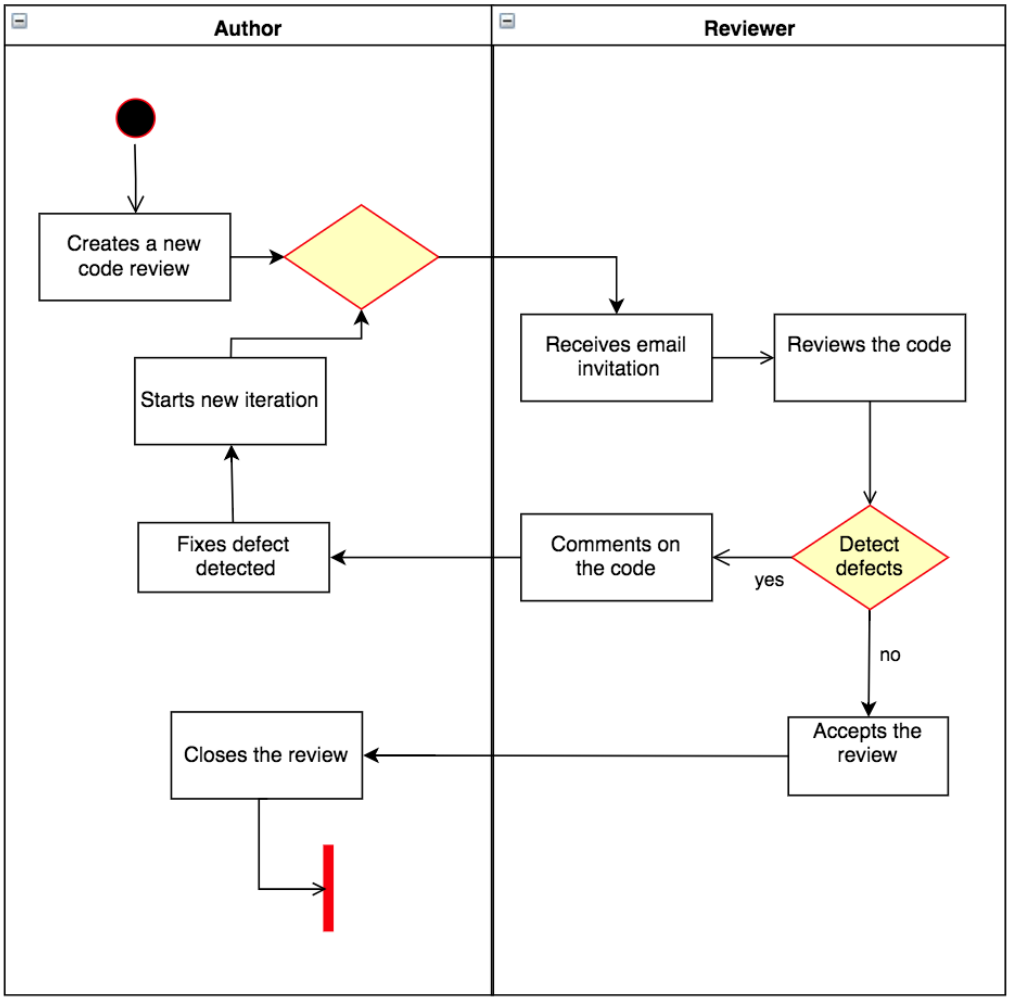


Figure 5.7: Swimlanes diagram representing the code review work-flow.

Once the author has corrected all defects identified, he begins a new iteration. By then, the reviewer has to check all the new changes and make sure they have been corrected in the right way.

In case the reviewer does not comply with the fixes, he can write new comments and a new iterations begins. This process is repeated until the reviewer finds no more defects.

If the reviewer believes the code's quality has reached the desired standards and all critical priority conversations have been marked as fixed, he will accept the review and the author can close it.

The goal of differentiating comments priority is to avoid forcing the same rigidity to all defects and let reviewers' feedback flow without compromising the author. Moreover, it is important because by letting the reviewer choose the importance of his feedback, the author will not be able to misinterpret it and the review can not be closed until all critical feedback has been taken care of.

Every time a code review is closed, an email notification (See figure 5.8) is sent to the participants containing all the related data (i.e., number of defects found, time elapsed). Such data is also stored in the system to provide overall statistics related to

users or related to issues.

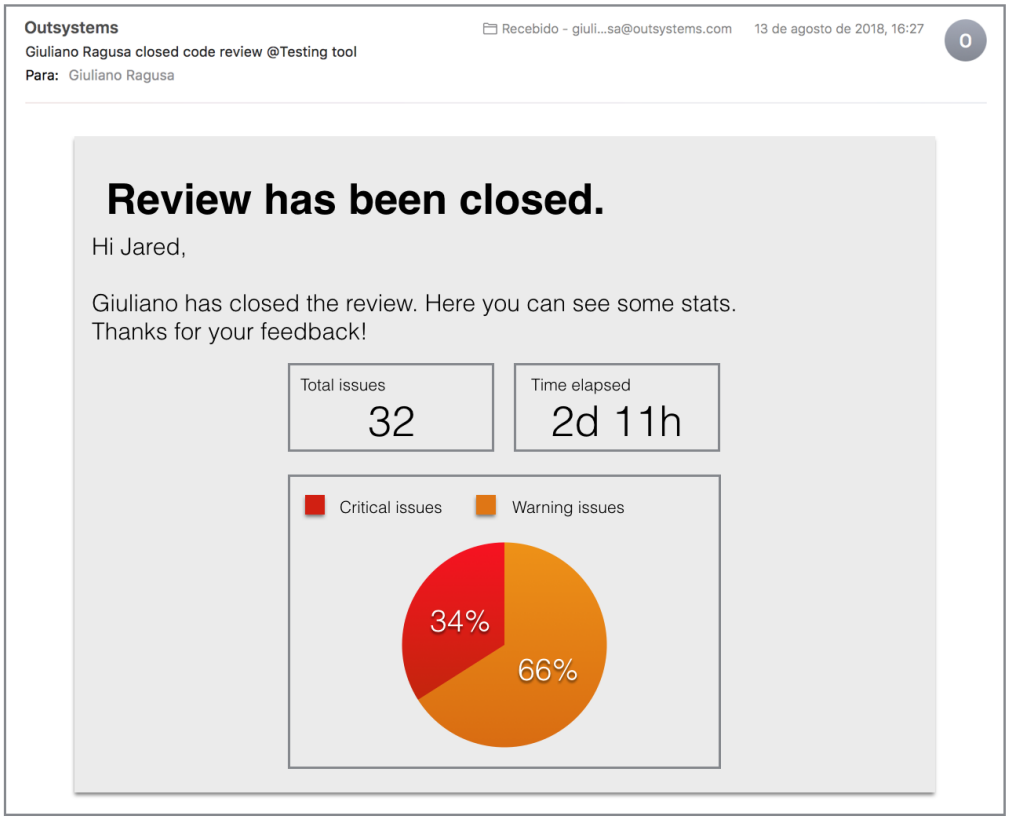


Figure 5.8: Email notification containing review's statistics

5.3 Participant selection

Choosing the right code reviewers is important to the overall performance of our solution, in fact, as important as any other main feature.

Always with the goal of easing the code review process, upon creating a new review, our tool always recommends selecting two reviewers. Such number justification is derived from Sauer *et al.* study [47] showing that two reviewers would be the optimal relation between the number of defects found and the number of reviewers. The number of additional defects detected did not justify the cost of adding other reviewers.

Moreover, and in accordance with Bosu [46], reviewers who had reviewed a file before were almost twice more useful. Therefore, in order to help the author sort the most valuable participants, the tool also orders automatically the available reviewers by their previous code review history on file.

5.4 Reading techniques

Even though reading techniques have proven to result in dramatic improvements in inspection effectiveness and efficiency [7], our process does not force any reading

technique. The main reason is that no specific reading technique would be able to fit most VPLs, and we aim to maintain a high degree of freedom during the code review process. Nonetheless, we strongly recommend future users of this tool to use the reading technique described in Section 2.3.1 based on the company, team or project requirements.

Chapter 6

Results

The metrics used in the analysis were gathered using usability experiments presented in Section 2.5. We used the same questions for both code review methods (current and new method).

The testers had to analyze real case code review scenarios and then answer the set of proposed questions. To maximize the authenticity of the results we used two sets of two artifacts with similar complexity, yet different. Between each tester, both the artifacts and the code review method were alternated to all the combinations possible. The two set of artifacts consisted of two artifacts developed by developers at OutSystems and two UML class diagrams.

A total of 30 testers participated in the experiment. The testers were all computer scientists, 60% of them were male and 40% female, they were all aged between 22 and 43 years old, and half of them already had previous experience with code review.

6.1 Usability experiment

To understand how the two different methods approach some important code review topics, we analyzed the scores gathered from a set of 8 questions. The questions had to be objective and the answers should be either be correct or not. Table 6.1 includes the list of question, detailed overview of the results and comments about the new code review method.

Table 6.1: Results of usability tests

Question	Correct Answer Rate (%)		New method comments
	Current method	New method	
1. Who is the author of the changes?	60%	100%	The participants did not have any trouble finding the information since it is now present on review panel.
2. Which are the changes between versions?	100%	100%	Same result as the original method.
3. Are the nodes changes clear?	30%	100%	The colored differences highlighted made the changes between revisions clear.
4. Is all information relevant?	33%	100%	The participants were not faced anymore with not pertinent information.
5. How can feedback be given?	20%	100%	Testers did not have difficulties understanding how to give their feedback to the author.
6. How do you know if the feedback is not redundant?	5%	95%	Flags on the code and the centralized feedback made possible to access previous reviewers' comments.
7. What is the feedback related to?	25%	88%	By having flags in the code redirecting to a specific comment, users had much less trouble relating feedback.
8. Can feedback be misinterpreted and the review closed without being fixed?	40%	90%	Authors felt sure they would not close a review by misinterpreting feedback since it was defined by the reviewer.

6.1.1 Descriptive statistics

Table 6.3 shows the descriptive statistics for the data collected from the questions answers. For this analysis each question had identical weight. The table contains statistics for the current and the new method of code review for VPLs.

Table 6.2: Descriptive statistics

	Method	N	Mean	Std. Deviation	Skewness	Kurtois	Shapiro-Wilk
Questions	Current	17	39.12	10.34	1.4	2.332	0.222
	New	13	96.62	1.78	-1.107	-0.591	0.004

In Figure 6.1 we can see an overview of the answers distribution. Each number present in the horizontal axis corresponds to a question from table 6.1.

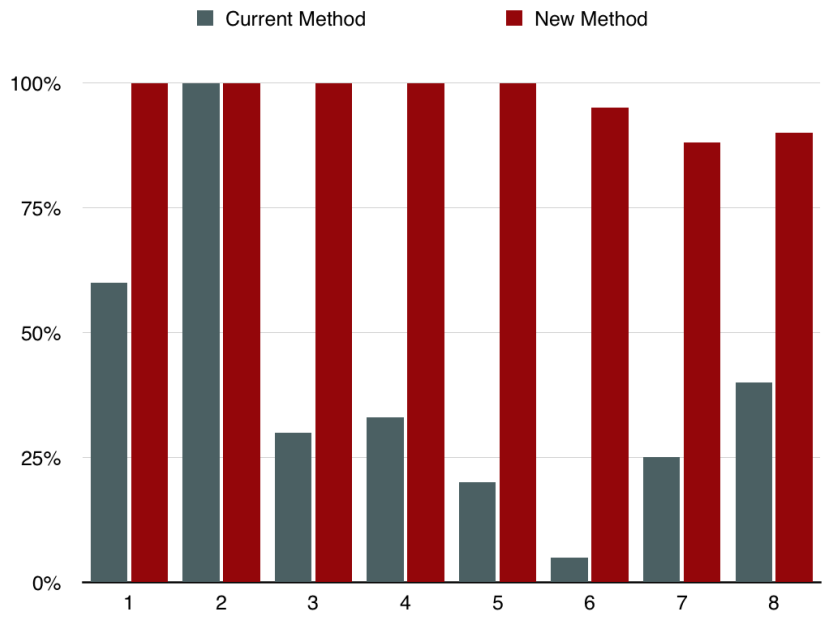


Figure 6.1: Graph of answer distribution

6.2 SUS and TLX

To determine the solution's usability and the work load we analysed the SUS and TLX scores gathered from usability experiments. By analysing these two scores we can determine whether the difference between the current and the new method are significant and relevant. As a reminder, a higher SUS score is better while a lower TLX score is better.

6.2.1 Descriptive statistics

Table 6.3 contains descriptive statistics for the SUS and TLX data collected while performing the usability tests. The table is grouped by score type and each type contains statistics for the current and the new method of code review for VPLs.

Table 6.3: Descriptive statistics

	Method	N	Mean	Std. Deviation	Skewness	Kurtois	Shapiro-Wilk
SUS	Current	17	35.55	9.87	0.659	0.255	0.931
	New	13	83.50	7.36	-0.297	-0.838	0.960
TLX	Current	17	35.30	6.06	-0.342	-0.087	0.686
	New	13	19.20	4.63	-0.813	0.382	0.245

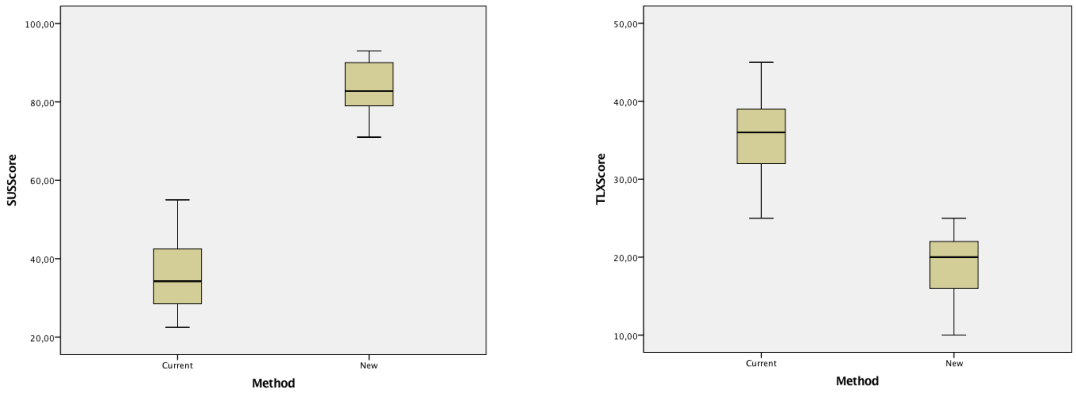
6.2.2 Hypotheses testing

Welch's t-test was used instead of the Student's t-test for testing the difference in SUS and TLX scores between the current and the new method of code review, since it has been shown that the Welch's t-test is better suited for studies with different sample sizes[80]. Table 6.4 contains: the means for both SUS and TLX; the difference between the current and the new method means; the 95% confidence interval of the difference; t, df and p-values.

Table 6.4: Welch's t-test scores

	Current method mean	New method mean	Difference	95% Dif. CI Lower	95% Dif. CI Upper	t	df	p-value
SUS	35.55	83.30	-47.75	-51.83	-37.22	-4.97	18.91	0.001
TLX	35.30	19.20	15.43	10.56	25.32	3.21	23.10	0.019

We hypothesized that the new method of code review would have a higher usability rating when compared to the current method. The SUS and TLX scores differed significantly according to Welch's t-test, $t_{SUS}(-4.97) = 18.91$, $p_{SUS} = .001$ and $t_{TLX}(3.21) = 23.10$, $p_{TLX} = .019$. On average, users who tested the current method of code review gave it a SUS score of 35.55 and a TLX score of 35.30, while users who tested the new method of code review gave it a SUS score of 83.38 and a TLX score of 19.20. The 95% confidence interval of the difference for the effect of the new method on the SUS score is between -51.83 and -37.22 and on the TLX score it is between 10.56 and 25.32. As such, these results support our hypothesis. The above is further shown in Figure 6.2 which contains a graphical display of the SUS and TLX scores.



(a) SUS score

(b) TLX raw

Figure 6.2: Box plot of distribution

6.3 Discussion of results and implications

Looking at the results achieved comparing the two methods through usability experiments (Table 6.1), it is clear that having all the features and mechanisms present in a single tool is important for the developers.

The results of the usability experiments also illustrate the importance of a well design user interface, which helps users feel more confident when using the tool. This corroborates Nielsen's usability heuristics [58]. In addition, other conclusions were also made: using the current method, authors are prone to misinterpret reviewers' feedback which can cause defective code to be merged into the main branch, impacting on the quality of the software produced; Beside, Bosu's analysis [46] was further confirmed when during the usability tests, reviewers with less code review experience tended to give less relevant feedback. This could suggest the need of always having at least one experienced reviewer within the participants; Finally, the nonexistence of automatic notifications in the current method resulted in potential forgotten reviews. Therefore, we can conclude that it is extremely important to continuously notify the participants of any update in the review.

When analyzing the SUS and TLX scores (Table 6.4 and Figure 6.2) the TLX score decreases as the SUS score increases. This seems to suggest that there is a direct relationship between the tool usability and the perceived workload. As seen in Subsection 6.2.2 the results support our hypothesis. As such, we conclude that the dissertations objectives were achieved.

6.4 Threats to validity

Although we chose two sets of similar complexity artifacts for the two different VPLs, we acknowledge possible threats to validity.

Of the threats presented by Wohlin et al. in [81], our concerns were mainly related to population and instruments selection. Due to resource constraints, the results were

obtained from a population of 30 developers chosen within OutSystems. This bias in the selection means that we did not randomly select a sample from the community of VPLs developers, neither had a large population. Besides, the similarity between the artifacts chosen may have caused the population to learn from the first scenario.

Ideally, the solution should be tested with more significant amounts of data and by users of a different background. This would allow us to measure the usability from different language viewpoints and also understand how other language developers with different perspectives felt using the tool.

6.5 Summary of results

The results obtained during the analysis clearly confirmed our hypothesis. However, besides proving that the new method of code review had higher usability rating and lower task loads then the current method, we also concluded that our tool helped to decrease the chances of bugs being introduced in the main branch. This was achievable by forcing a non-closure policy if the detected defects were not yet fixed. Moreover, the results indicated that to maximize the relevant feedback, at least one reviewer should be experienced. The last assumption withdrawn from the results was is extremely important to keep all participants updated on the review changes. Thus, shortening the code review cycles.

Chapter 7

Conclusions and future work

This dissertation main objective was to develop a tool to make the code review task on visual programming languages easier. This tool, together with a systematic process would allow to deliver higher quality products in a shorter amount of time.

We first started this project by performing interviews to understand the current need of VPLs developers. Also, we benchmarked the current software industry to extract leading practices regarding code review. With all the data gathered we were able to create mock-ups of the interface. After several iterations between: usability test and fixing issues reported, we ended with a solid and compliant base to start developing the actual solution.

The next step was developing the tool and creating a process that could be applied to most VPLs. Once done, to evaluate the performance of this new code review methodology we relied on a predefined evaluation process and had two sets of testers.

The first was faced with two OutSystems artifacts and the two different methods of code review. The different method to use was, alternately, the current method of code reviewing VPLs and the new solution proposed. We then conducted usability experiments where the testers were asked to perform the code review on each scenario and then asked a series of questions related to the methodology being use. At the end they were asked to answer a series of predefined questions to score the solution in usability and workload scales.

The second set of testers would go trough the same procedure but this time reviewing class diagrams instead of OutSystems code.

With all the collected data the evaluator should then decide whether the proposed solution needs to be improved, if it does not, then the process ends. Otherwise, the evaluator should then use all the relevant feedback to improve the existing solution. The process should then restart using the new version.

The comparison analysis between and current and new method of reviewing visual programming languages confirmed that the process is effective, that the new mechanism has a higher usability rating and lower workload levels. But, other conclusions were also made: Misinterpreted feedback has a large impact on the quality of the software produced; users with less code review experience tend have to less relevant

feedback (which goes in line with Bosu's analysis [46]); and it is extremely important to maintain both reviewer and author updated of any change in the review.

7.1 Contributions

The main contribution of this dissertation was new method of the code review developed for VPLs which performed significantly better, with a much higher usability rating and lower work-load. This was achieved by applying the proposed process, which also serves to demonstrate the effectiveness of the process.

Another contribution came out to a showpiece paper (See appendix A) accepted in IEEE Symposium on Visual Languages and Human-Centric Computing 2018 [82]. Alongside, a presentation and demonstration of the tool to the VL/HCC community at the conference venue.

7.2 Future work

This dissertation main goal wast to create method of code review, where a tool and a process were proposed to improve the code review on VPLs. Even though these objective was achieved, it can still be improved. The tool can be expanded with further features and new language converters to be applicable to more visual languages. Such features would benefit both the tool and its users. These include:

Code linting. Support automatic code checking. This feature, will allow user to add rules using a Domain specific language (DSL). The tool will then will run the entire code searching for vulnerabilities and bad practices that go against those predefined rules. All divergences will be highlighted in the diff-viewer and the problems' detailed explanation will be given. For example, if a node has not been labeled or there is more than one node with the same label. The feature can be useful since some bad practices can go unnoticed even after the code review. However, the point is not to slow down the entire code review process, forcing the author to solve all divergences. All the conflicts will appear as warnings which do not prevent the author from closing the review.

Automatic reviewers selection. Another feature to offer is reviewers recommendation based on their expertise. Finding a suitable reviewer can be difficult and, delayed or forgotten reviews are the consequence. Automating reviewer selection with algorithms can mitigate this problem. The tool relies on algorithms to evaluate the projects review history and calculate the review expertise of all potential reviewers for a changes as the foundation for the recommendation.

Wider range of support. Even today's simplest application contains a diversity of different layers. From data model, work-flow processes, business logic to user interfaces and all of them would benefit from code review. In the current version of the system, code review is available for OutSystems' logic and processes and UML Class Diagrams. However, the objective is to expand the tool's support to allow to review UI, data model and other visual programming languages.

Finally, all these features would require more testing to make sure the process does not become cumbersome. The main idea is to always maintain high usability scores and decreasing the workload rates.

Bibliography

[1] B. Jost, M. Ketterl, R. Budde, and T. Leimbach. "Graphical Programming Environments for Educational Robots: Open Roberta - Yet Another One?" In: *2014 IEEE International Symposium on Multimedia*. 2014, pp. 381–386. doi: 10.1109/ISM.2014.24.

[2] R. Dehouck. *The maturity of visual programming*. 2015. url: http://www.craft.ai/blog/the-maturity-of-visual-programming/ (visited on 01/10/2018).

[3] J.-M. Sáez-López, M. Román-González, and E. Vázquez-Cano. "Visual programming languages integrated across the curriculum in elementary school: A two year case study using "Scratch" in five schools." In: *Computers & Education* 97 (2016), pp. 129 –141. issn: 0360-1315. doi: https://doi.org/10.1016/j.compedu.2016.03.003. url: http://www.sciencedirect.com/science/article/pii/S0360131516300549.

[4] M. Revell. *What is visual programming*. 2017. url: https://www.outsystems.com/blog/what-is-visual-programming.html (visited on 01/10/2018).

[5] Smartbear. *Why Code Review*. 2017. url: https://smartbear.com/learn/code-review/why-review-code/ (visited on 12/02/2017).

[6] S. Lammers. *Programmers at Work: Interviews With 19 Programmers Who Shaped the Computer Industry*. Tempus Series. Tempus Books of Microsoft Press, 1986. isbn: 9781556152115. url: https://books.google.pt/books?id=C31GAAAAYAAJ.

[7] O. Laitenberger, C. Atkinson, M. Schlich, and K. El Emam. "An experimental comparison of reading techniques for defect detection in UML design documents." In: *Journal of Systems and Software* 53.2 (2000), pp. 183–204.

[8] Outsystems. *OutByNumbers - Benchmark Overview Report*. Tech. rep. OutSystems, 2013. url: http://www.outsystems.com/res/OutbyNumbers-DataSheet.pdf.

[9] OutSystems. *OutSystems Platform - Architecture and Infrastructure Overview*. Tech. rep. OutSystems, 2013. url: https://www.outsystems.com/home/documentdownload/178/8/0/0.

[10] B. Grácio. "Agregado: Compilar Sistemas NoSQL na Plataforma OutSystems." In: Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa: MA thesis, 2015.

[11] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2008. url: https://www.w3.org/TR/REC-xml/ (visited on 12/20/2017).

[12] Microsoft. *Windows Server 2016*. 2016. url: https://www.microsoft.com/en-us/cloud-platform/windows-server (visited on 11/29/2017).

[13] Microsoft. *Windows SQL server*. 2017. url: https://www.microsoft.com/en-us/sql-server/sql-server-2017 (visited on 11/29/2017).

[14] Microsoft. *ASP.NET*. 2017. url: http://www.asp.net/ (visited on 11/29/2017).

[15] Docker. *What is a Container*. 2017. url: https://www.docker.com/what-container (visited on 12/20/2017).

[16] Microsoft. *IIS*. 2016. url: https://www.iis.net/ (visited on 11/29/2017).

[17] Oracle. *Oracle*. 2016. url: http://www.oracle.com/index.html (visited on 11/29/2017).

[18] Oracle. *MySQL*. 2016. url: https://www.mysql.com/ (visited on 11/29/2017).

[19] M. Fagan. "A history of software inspections." In: *Software pioneers*. Springer, 2002, pp. 562–573.

[20] M. E. Fagan. "Design and code inspections to reduce errors in program development." In: *IBM Systems Journal* 38.2/3 (1999), p. 258.

[21] P. C. Rigby and D. M. German. *A preliminary examination of code review processes in open source projects*. Tech. rep. Technical Report DCS-305-IR, University of Victoria, 2006.

[22] T. Baum, F. Kortum, K. Schneider, A. Brack, and J. Schauder. "Comparing pre-commit reviews and post-commit reviews using process simulation." In: *Journal of Software: Evolution and Process* (2017).

[23] G. H. Travassos. "Software Defects: Stay Away from Them. Do Inspections!" In: *2014 9th International Conference on the Quality of Information and Communications Technology*. 2014, pp. 1–7. doi: 10.1109/QUATIC.2014.8.

[24] D. A. Wheeler, B. Brykczynski, and R. N. Meeson Jr., eds. *Software Inspection: An Industry Best Practice for Defect Detection and Removal*. 1st. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996. isbn: 0818673400.

[25] R. B. Grady and T. V. Slack. "Key lessons in achieving widespread inspection use." In: *IEEE software* 11.4 (1994), pp. 46–57.

[26] J. Barnard and A. Price. "Managing code inspection information." In: *IEEE software* 11.2 (1994), pp. 59–69.

[27] E. Kantorowitz, A. Guttman, and L. Arzi. "The performance of the N-Fold requirement inspection method." In: *Requirements Engineering* 2.3 (1997), pp. 152–164.

[28] E. F. Weller. "Lessons from three years of inspection data (software development)." In: *IEEE software* 10.5 (1993), pp. 38–45.

[29] J. C. Kelly, J. S. Sherif, and J. Hops. "An analysis of defect densities found during software inspections." In: *Journal of Systems and Software* 17.2 (1992), pp. 111–117.

[30] A. Bacchelli and C. Bird. "Expectations, outcomes, and challenges of modern code review." In: *Proceedings of the 2013 international conference on software engineering*. IEEE Press. 2013, pp. 712–721.

[31] P. C. Rigby and C. Bird. “Convergent contemporary software peer review practices.” In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 202–212.

[32] J Lahtinen. *Application of the perspective-based reading technique in the nuclear I and C context. CORSICA work report 2011*. Tech. rep. VTT Technical Research Centre of Finland, 2012.

[33] Wikipedia. *Ad-hoc*. 2017. url: https://en.wikipedia.org/wiki/Ad_hoc (visited on 12/24/2017).

[34] O. S. Akinola and A. O. Osofisan. “An Empirical Comparative Study of Checklist based and Ad Hoc Code Reading Techniques in a Distributed Groupware Environment.” In: *arXiv preprint arXiv:0909.4260* (2009).

[35] O. Laitenberger. “A survey of software inspection technologies.” In: *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*. World Scientific, 2002, pp. 517–555.

[36] T. Gilb, D. Graham, and S. Finzi. *Software inspection*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[37] D. L. Parnas and D. M. Weiss. “Active design reviews: principles and particles.” In: *The Journal of systems and software* 7.4 (1987), pp. 259–265.

[38] G. S.L.O.L.F.S.F.S. S. Basili V. R. and M Zelkowitz. *Lab package for the empirical investigation of perspective-based reading*. 2011. url: http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/manual.html (visited on 12/07/2017).

[39] O. Laitenberger. “Cost-effective detection of software defects through perspective-based inspections.” In: *Empirical Software Engineering* 6.1 (2001), pp. 81–84.

[40] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. V. Zelkowitz. “The empirical investigation of perspective-based reading.” In: *Empirical Software Engineering* 1.2 (1996), pp. 133–164.

[41] M. E. Fagan. “Advances in software inspections.” In: *Pioneers and Their Contributions to Software Engineering*. Springer, 2001, pp. 335–360.

[42] P. M. Karl Wiegers. *Lightweight Tool Support For Effective Code Reviews*.

[43] Y. Legaspi. *How to prepare for effective code reviews: catch more bugs with these best practices*. 2017. url: https://raygun.com/blog/top-3-tips-for-delightful-code-reviews/ (visited on 12/18/2017).

[44] J. Carver. “The impact of background and experience on software inspections.” In: *Empirical Software Engineering* 9.3 (2004), pp. 259–262.

[45] P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German. “Contemporary peer review in action: Lessons from open source development.” In: *IEEE software* 29.6 (2012), pp. 56–61.

[46] A. Bosu, M. Greiler, and C. Bird. “Characteristics of useful code reviews: An empirical study at microsoft.” In: *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE. 2015, pp. 146–156.

[47] C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. “The effectiveness of software development technical reviews: A behaviorally motivated program of research.” In: *IEEE Transactions on Software Engineering* 26.1 (2000), pp. 1–14.

[48] C. Hannebauer, M. Patalas, S. Stünkelt, and V. Gruhn. “Automatically recommending code reviewers based on their expertise: An empirical comparison.” In: *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE. 2016, pp. 99–110.

[49] M. B. Zanjani, H. Kagdi, and C. Bird. “Automatically recommending peer reviewers in modern code review.” In: *IEEE Transactions on Software Engineering* 42.6 (2016), pp. 530–543.

[50] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. “Who should review my code? A file location-based code-reviewer recommendation approach for modern code review.” In: *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE. 2015, pp. 141–150.

[51] H. Kagdi, M. Hammad, and J. I. Maletic. “Who can help me with this source code change?” In: *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE. 2008, pp. 157–166.

[52] K. Charmaz. “Constructing grounded theory: A practical guide through qualitative research.” In: *SagePublications Ltd, London* (2006).

[53] B. G. Glaser. *Advances in the methodology of grounded theory: Theoretical sensitivity*. 1978.

[54] A. L. Strauss. *Qualitative analysis for social scientists*. Cambridge University Press, 1987.

[55] K. Charmaz and L. L. Belgrave. *Grounded theory*. Wiley Online Library, 2007.

[56] R. Molich and J. Nielsen. “Improving a human-computer dialogue.” In: *Communications of the ACM* 33.3 (1990), pp. 338–348.

[57] J. Nielsen. *Ten usability heuristic*. 2005. url: https://www.nngroup.com/articles/ten-usability-heuristics/ (visited on 01/20/2018).

[58] J. Nielsen. “Enhancing the explanatory power of usability heuristics.” In: *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM. 1994, pp. 152–158.

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

[60] A. Bangor, P. T. Kortum, and J. T. Miller. “An empirical evaluation of the system usability scale.” In: *Intl. Journal of Human–Computer Interaction* 24.6 (2008), pp. 574–594.

[61] S. Rubio, E. Díaz, J. Martín, and J. M. Puente. “Evaluation of subjective mental workload: A comparison of SWAT, NASA-TLX, and workload profile methods.” In: *Applied Psychology* 53.1 (2004), pp. 61–86.

[62] Y. Liu and C. D. Wickens. “Mental workload and cognitive task automaticity: an evaluation of subjective and time estimation metrics.” In: *Ergonomics* 37.11 (1994), pp. 1843–1854.

[63] S. G. Hart. “NASA-task load index (NASA-TLX); 20 years later.” In: *Proceedings of the human factors and ergonomics society annual meeting*. Vol. 50. 9. Sage Publications Sage CA: Los Angeles, CA. 2006, pp. 904–908.

[64] S. G. Hart and L. E. Staveland. “Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research.” In: *Advances in psychology*. Vol. 52. Elsevier, 1988, pp. 139–183.

[65] N. Human Performance Research Group et al. “NASA Task Load Index (TLX) v. 1.0: Paper and Pencil Package.” In: *Moffett Field, CA: NASA Ames Research Center* (1986).

[66] E. A. Bustamante and R. D. Spain. “Measurement invariance of the Nasa TLX.” In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. Vol. 52. 19. SAGE Publications Sage CA: Los Angeles, CA. 2008, pp. 1522–1526.

[67] Microsoft. *Microsoft*. 2017. url: https://www.microsoft.com/pt-pt/store/b/home?invsrc=search&OCID=AID620866_SEM_Weh9pwAAAHuNr3wI:20171213110442:s&s_kwcid=AL!4249!3!222300191708!e!!g!!microsoft&ef_id=Weh9pwAAAHuNr3wI:20171213110442:s (visited on 12/05/2017).

[68] Gerrit. *Gerrit Code Review*. 2017. url: https://www.gerritcodereview.com/ (visited on 12/05/2017).

[69] Phacility. *Phabricator*. 2017. url: https://www.phacility.com/phabricator/ (visited on 12/05/2017).

[70] Reviewboard. *Reviewboard*. 2017. url: https://www.reviewboard.org/ (visited on 12/05/2017).

[71] Cisco. *Cisco*. 2017. url: https://www.cisco.com/ (visited on 12/05/2017).

[72] Smartbear. *CodeCollaborator*. 2017. url: https://smartbear.com/product/collaborator/overview/ (visited on 12/05/2017).

[73] J. Cohen, E. Brown, B. DuRette, and S. Teleki. *Best kept secrets of peer code review*. Smart Bear, 2006.

[74] Smartbear. *Smartbear*. 2017. url: https://smartbear.com/product/collaborator/industries/automotive/ (visited on 12/05/2017).

[75] F. Henderson. “Software Engineering at Google.” In: *CoRR* abs/1702.01715 (2017). arXiv: 1702.01715. url: http://arxiv.org/abs/1702.01715.

[76] JetBrains. *Upsource*. 2017. url: https://www.jetbrains.com/upsource/ (visited on 12/28/2017).

[77] L. MacLeod, M. Greiler, M. A. Storey, C. Bird, and J. Czerwonka. “Code Reviewing in the Trenches: Understanding Challenges and Best Practices.” In: *IEEE Software* PP.99 (2017), pp. 1–1. issn: 0740-7459. doi: 10.1109/MS.2017.265100500.

[78] R. C. Camp. “Benchmarking: the search for industry best practices that lead to superior performance.” In: *Benchmarking: the search for industry best practices that lead to superior performance*. ASQC/Quality Resources, 1989.

[79] Sketch. *Sketch*. 2018. url: https://www.sketchapp.com/ (visited on 02/02/2018).

[80] B. Kitchenham, L. Madeyski, D. Budgen, J. Keung, P. Brereton, S. Charters, S. Gibbs, and A. Pohthong. “Robust statistical methods for empirical software engineering.” In: *Empirical Software Engineering* 22.2 (2017), pp. 579–630.

[81] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[82] G. Ragusa and H. Henriques. “Code review for visual programming languages.” In: *IEEE Symposium on Visual Languages and Human-Centric Computing 2018*. IEEE. 2018, pp. 287–288.

Appendix

A

Paper

Code review tool for Visual Programming Languages

Giuliano Ragusa
FCT/UNL
Almada, Portugal
g.ragusa@campus.fct.unl.pt

Henrique Henriques
OutSystems SA
Linda-a-Velha, Portugal
henrique.henriques@outsystems.com

***Abstract*—Code review is a common practice in the software industry, in contexts spanning from open to close source, and from free to proprietary software. Modern code reviews are essentially conducted using cloud-based dedicated tools. Existing review tools focus in textual code. In contrast, support of low-code software languages, namely Visual Programming Languages (VPLs), is not readily available. This presents a challenge for the effectiveness of the review process with a VPL. This showpiece will present VPLreviewer, a code review tool for VPLs. VPLreviewer provides a wide range of mechanisms previously not available to a VPL. It is expected to improve of communication among the stakeholders who have to review artifacts constructed with VPLs, with mechanisms that are easy to learn, use and understand.**

I. Introduction

Code review's importance has already been proven. Several researchers provided evidence on traditional code inspection's benefits, especially in terms of defect finding [1], [2], [4]. And although finding bugs is important, the foremost reason for introducing code review in big companies, such as Google, is to improve code understandability and maintainability [3].

Visual Programming Languages (VPLs) have substantially increased their presence in the software industry in recent years. Yet, mechanisms to help increase software quality have not evolved accordingly. Thus, visual artifacts are not supported by existing textual programming languages code review tools, hampering the code review process.

OutSystems was used as a case study for VPLreviewer. The company prodives a low-code platform that allows to visually develop entire applications. Even tough at OutSystems code review is a concern, the lack of tools for VPLs makes reviewing them a major problem. The code review still happens and is encouraged. However, it isn't productive and has a negative impact on delivery speed.

II. The tool

We developed a web-based tool to assist the code reviews of VPLs. The tool contains a wide range of features inspired in both already existent software for textual code review and problems identified in a visual programming context at OutSystems, such as: not being able to comment on a specific change, not knowing the context or the order of the changes, etc.

VPLreviewer is a generic code review tool for VPLs. Instead of directly consuming the artifact from a specific visual language, we opted to feed the tool with a specific JSON structure containing all information of an artifact. Although the first plugin was developed for OutSystems, different plugins can be easily created to support other languages (e.g. Petri nets).

The solution's architecture encapsulates together two main components, making them work as one unique body from an outsider perspective. The architecture is split to allow the logic to be refactored or replaced without it impacting the UI.

These components are:

- **Core services.** Represents the back-end of the tool. They are responsible for converting and storing the visual artifacts into tool-readable structures. This pre-processing enables the manipulation of each of the visual elements.
- **Tool.** The tool is able to compute differences between versions and present them side-by-side in a visual manner, along with all features designed to help participants conduct code reviews.

The user interface is the only component end users have access to. With its straightforward design and intuitive features, participants can focus their attention on detecting defects and consequently increasing code quality.

Figure 1 shows the tool's interface, while reviewing a visual artifact. All tool features were inspired by successful implementations of code review tools and from issues identified upon interviewing low-code developers at OutSystems. These include:

- **Diff-viewer** a side-by-side view (A) of two different versions of the same artifact;
- **Changes highlight** different color highlighting to changes for quickly understanding what changed. Red for deleted elements, yellow for modified and green for added;
- **Threads of discussion** participants can select a single or multiple nodes and create new discussion threads (B);
- **Issue Tracking Software integration** JIRA Atlassian integration for more extended context on change (C);
- **Properties** all changes have a properties table (D) to more in-depth context of a specific change;

978-1-5386-4235-1/18/$31.00

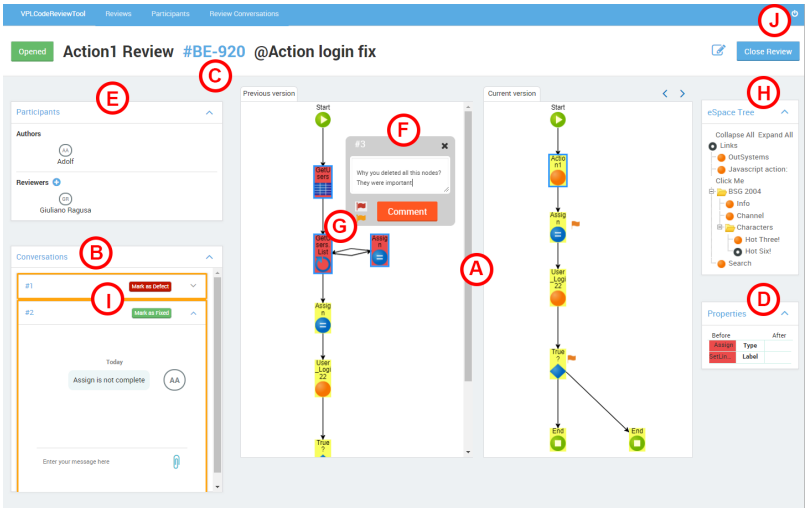


Fig. 1. Tool's user interface (source at https://goo.gl/i9VHd4)

- **Files list** Tree listing of all files (H) related with the review;
- **Participants list** Ability of adding participants (E) as needed;
- **Notifications** Automatic notifications to every participant whenever a review is updated;
- **Data** stores data of each review and provides metrics such as number of defects found, average time per review, etc.

The tool presents a side-by-side view with all the differences highlighted. The reviewer will be able to open threads of conversations by simply right clicking (F) on any node and all threads opened will be listed on the side. The participant can chose the importance of his comment by switching the thread's flag (G) from warning (orange) to critical (red). By clicking any highlighted change, the reviewer will be able to get all the information on that specific change, giving him a more accurate context. If needed, the reviewer can click on the issue's hyperlink which will redirect him directly to the company's issue tracking platform. Reviews are only closed (J) when they need no more iterations and all critical threads of conversations are marked as fixed (I).

All in all, our main objective with this tool is to support, but not restrict the code review process, providing a flexible and lightweight tool. Our tool minimizes the effort devoted to administrative aspects such as scheduling meetings, encouraging attendance, and recording review comments. Using VPLreviewer makes it effortless to invite reviewers, distribute artifacts and gather feedback. Instead of recording issues on separate log forms, the tool lets reviewers insert their comments in context, right next to the visual code element in question, facilitating discussions among the reviewers on issues that are brought up.

III. Impact for the VL/HCC community

VPLreviewer is of interest to the VL/HCC community because, even tough VPLs have been rapidly evolving, code review software has stagnated on textual languages. Our work is an attempt to provide support for the code review process and increase the quality of the software developed with VPLs.

We plan to release VPLreviewer to the VPLs industry in the near future. Therefore, we would benefit from demonstrating the tool during the VL/HCC showpiece presentation session by allowing a community of experts to go through the tool and give us feedback to improve it before its release.

IV. Presentation

The approach and tool presented by this showpiece paper will be further demonstrated using a screen-cast video (available at https://youtu.be/wgnZ_c235NQ). An author will explain the tool features and how the code review process is conducted. In addition to the video, attendees will also be able to test the tool with a variety of visual artifacts, in both author and reviewer perspective.

V. Acknowledgments

The authors would like to thank OutSystems for all support given throughout this project.

References

[1] Fagan, Michael. "Design and code inspections to reduce errors in program development." Software pioneers. Springer, Berlin, Heidelberg, 2002. 575-607.

[2] Laitenberger, Oliver, et al. "An experimental comparison of reading techniques for defect detection in UML design documents." Journal of Systems and Software 53.2 (2000): 183-204.

[3] Sadowski, Caitlin, et al. "Modern code review: a case study at google." Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. ACM, 2018.

[4] Bacchelli, Alberto, and Christian Bird. "Expectations, outcomes, and challenges of modern code review." Proceedings of the 2013 international conference on software engineering. IEEE Press, 2013.

Annex I

Interview's topics

Presented here are the topics addressed during the interviews.

1. What is your role in the company?
2. What do you understand by code review?
3. What is the life cycle of the code?
4. Do you perform code reviews?
    a) If not:
        i. Why not? No need/ no tools /no time?
        ii. You do not feel the need of reviewing the code?
        iii. What could be helpful for code review?
    b) If yes:
        i. How do you do it?
        ii. How do you feel doing code review?
        iii. Do you use any tools?
        iv. How do you select participants?
        v. Do you know the optimal number of reviewers?
        vi. Do newbies have any kind of guidance? Do you apply any specific reading technique?
        vii. Which are the drawbacks of your current technique? What could help you in the process? Both for the author and the reviewers