

Towards the Systematic Testing of Virtual Reality Programs

Stevão A. Andrade

Universidade de São Paulo - ICMC

São Carlos, Brasil

stevao@icmc.usp.br

Fatima L. S. Nunes

Universidade de São Paulo - EACH

São Paulo, Brasil

fatima.nunes@usp.br

Márcio E. Delamaro

Universidade de São Paulo - ICMC

São Carlos, Brasil

delamaro@icmc.usp.br

Abstract—Background: Software testing is a critical activity to ensure that software complies with its specification. However, current software testing activities tend not to be completely effective when applied in specific software domains in Virtual Reality (VR) that has several new types of features such as images, sounds, videos, and differentiated interaction, which can become sources of new kinds of faults.

Aims: This paper presents an overview of the main VR characteristics that can have an impact on verification, validation, and testing (VV&T). Furthermore, it analyzes some of the most successful VR open-source projects to draw a picture concerning the danger of the lack of software testing activities.

Method: We compared the current state of software testing practice in open-source VR projects and evaluate how the lack of testing can be damaging to the development of a product. We assessed the incidence of code smells and verified how such projects behave concerning the tendency to present faults. We also perform the same analyses on projects that are not VR related to have a better understanding of these results.

Results: The results showed that the practice of software testing is not yet widespread in the development of VR applications. It was also found that there is a high incidence of code smells in VR projects. Analyzing Non-VR projects we noticed that classes that have test cases tend to produce fewer smells compared to classes that were not tested. Regarding fault-proneness analysis, we used an unsupervised approach to VR and Non-VR projects. Results showed that about 12.2% of the classes analyzed in VR projects are fault-prone, while Non-VR projects presented a lower fault-proneness rate (8.9%).

Conclusions: Regarding the application of software testing techniques on VR projects, it was observed that only a small number of projects are concerned about developing test cases for VR projects, perhaps because we still do not have the necessary tools to help in this direction. Concerning smells, we concluded that there is a high incidence in VR projects, especially regarding implementing smells and this high incidence can have a significant influence on faults. Finally, the study related to fault proneness pointed out that the lack of software testing activity is a significant risk to the success of the projects.

Index Terms—software testing, virtual reality, validation, code smells, fault proneness

I. INTRODUCTION

Technological advancement has led to the development of systems with new features such as images, sounds, videos and differentiated interaction. Thus, technologies such as Virtual Reality (VR) have led to possibilities of creating three-dimensional environments with real-time interaction.

Despite the great benefits of adopting VR for the development of applications in various areas, it poses new challenges

for verification, validation, and testing (VV&T) activities. For example, VR software presents original software structures, such as scene graphs, which may represent new sources of defects for programs. These new challenges motivated the development of some approaches that aim to contribute to the quality assurance process of software in the context of VR.

As mentioned by Corrêa et al. [1], there is interest in the literature on the subject. However, there is still no concept regarding systematized practices for conducting this activity. Studies have shown that the major problem remains in the difficulty to deal with test oracles, which is considered to be an open-ended research problem.

In general, the test activities for the VR domain are manually performed and mostly conducted only after the end of the development phase [2]. Such events generally support the generation of test requirements (functional and non-functional) which must be guaranteed before the product is delivered. The lack of studies that evaluate the cost of developing new techniques or using existing ones assess their effectiveness or even propose tools that can support their application, thus contributing to impact VV&T activities in general and aggravate this scenario.

Regardless of the programming technology used, a primary development goal is to produce high-quality software. Consequently, VR also needs to be tested and vetted for quality. Key questions related to quality include: *What should be tested?*, *What does “adequate testing” mean?*, *What is a failure in VR software?*

Systematic testing of the VR system must be based on fault models that reflect the structural and behavioral characteristics of VR software. Criteria and strategies for testing VR should be developed based on that fault model.

In this paper, we discuss whether new challenges for VV&T of VR exist that require novel techniques and methods, or instead, we need new ways of combining and using existing approaches. We also try to evaluate how much the lack of VV&T activities can negatively impact VR software development. To do so, we analyze the most popular open-source projects and categorize fault-proneness codes that could be mitigated by adopting VV&T activities.

This paper is organized as follows: Section II discusses the critical questions described above, as well as what testing approaches proposed in other domains could be reused for the

VR domain; Section III presents an exploratory study to assess how much the lack of VV&T activities can be prejudicial to open-source projects; Section IV discusses the results of the experiment presented; Section V points out some limitations related to this study; Section VI discusses related work and the conclusions and future work are shown in Section VII.

II. CHALLENGES AND ISSUES

Despite the benefits of adopting VR for the development of applications in several areas, this poses new challenges for software quality assurance activities. For example, software developed for the context of VR has unique software structures, which may represent new sources of faults for the programs developed [3]. These new challenges have motivated the development of some approaches that aim to contribute to the quality assurance process of software in the context of VR.

Automating software testing activities is often a complicated and challenging process. The main tasks of this activity include organizing, executing, registering the execution of the test cases and verifying the result of their execution.

In order to address these tasks, in the context of VR, some key points discussed in the next subsections should be understood.

A. What should be tested?

Virtual reality systems use individual hardware devices to allow the interaction with the user and the system. The work of graphics engines is not the primary concern for VR application developers. Defining scene graphs for organizing 3D objects in a VR world, managing virtual users, controlling sensors for detecting events such as object collision and processing events for reacting to user inputs are some of the typical elements of VR systems that the developers should be concerned about [4].

By observing the organization of 3D object elements and assets in scene graphs, it seems that it needs a higher-level type of test. In general, because they are independent, they do not have an architecture correlation of the source code. Therefore integration testing tends to be a more appropriate approach to be used. In integration testing, the main aim is to verify the communication between the units that make up the system.

B. What does “adequate testing” mean?

The solution to define this question: “*What does adequate testing mean?*” is to apply test criteria, which consists of a set of rules for dividing and evaluating the valid input domain for the program being tested. A test criterion defines elements of a program that must be exercised during its execution, thereby guiding the tester in the process of designing the test cases for the system. A test requirement may be, for example, a software-specific execution path, a functionality obtained through specification, a mutation-based approach, etc [5].

Corrêa et al. [1] presented a set of studies that deal with the application of software testing techniques to programs in the VR context, showing that there is an interest in the literature

on the subject, however, there is still no concept regarding systematized practices for the activity.

Due to the lack of defined requirements, it is not easy to identify test adequacy criteria for VR systems. How can we decide that testing is enough? This question needs to be adapted to the context.

C. What is a failure in a VR software?

In the context of VR applications, the testing activity hinges on the difficulty of systematizing how the behavior of a test case can be measured. This difficulty is described in the literature as “test oracle problems” and it appears in cases where traditional means of measuring the execution of a test case are impractical or are of little use in judging the correction of outputs generated from the input domain data [6], [7].

D. Can we reuse something?

General tools such as capturing and replaying can be used, but they offer a shallow level of abstraction. Thus, any small change to the system will result in the fact that the tests should be redone [8]. Therefore, using capture and replay tools cannot be used when the system is in development.

From a unit test point of view, we can still reuse a traditional approach in which we can quickly gauge the expected output to a method execution, ensuring that the smallest units of the VR system have been sufficiently tested against their specifications.

Regarding integration testing which is expected to handle new kinds of elements (3D objects, assets, behaviors, etc.) the literature review shows that we still need better-systematized practices for this activity [1].

E. What is done nowadays?

Almost all 3D applications require some common features. Therefore, developers tend to use platforms that provide these features out-of-the box. Using game engines is one of the most popular techniques among developers due to the fact it helps produce the systems, besides speeding up the development process.

Recently popular game engines, such as *Unity3D*¹ and *Unreal Engine*², released their own set of testing tools, which allows developers to produce automated testing during the development phase of the system which can substantially increase the stability of the product developed. Despite this, these tools still do not provide observable test criteria which is additionally repeatable, documented and does not rely on the tester’s creativity.

III. DO WE REALLY NEED TO TEST VIRTUAL REALITY SOFTWARE?

Considering popularizing VR application development, we are interested in understanding, from the software engineering point of view, how the development process of these applications is currently conducted. We are especially interested in

¹<https://docs.unity3d.com/Manual/testing-editortestrunner.html>

²<https://docs.unrealengine.com/latest/INT/Programming/Automation>

software testing practices in the development process of such applications in order to address what kinds of malfunctions the lack of test practice can lead to.

One of the most used approaches to quantify quality attributes in software projects is the evaluation of source code metrics. Source code metrics are a significant component for the software measurement process and are commonly used to measure fault proneness and improve the quality of the source code itself [9].

Another factor that can be exploited to evaluate code quality is to identify anti-patterns since some studies show that there is a correlation with fault proneness [10]. Therefore, these are two aspects that are taken into account in the evaluation carried out in our study to investigate the quality aspects of the code in the context of software testing.

Ghrairi et al. [11] made an exploratory study on *Github* and *Stack Overflow* in order to investigate which are the most popular languages and engines used in VR projects. According to their results, the most popular language for VR development is C#, and Unity is the most used game engine during VR application development. Thus, we focus our analyses targeting these characteristics.

A. Overview of the study

We formulated the following research questions regarding the quality analysis goal of VR projects.

- **RQ₁** : “*How does testing happen in open-source VR software systems?*” We focus on understanding how testing practices are being applied in open-source VR projects.
- **RQ₂** : “*What are the distribution of architecture, implementation and design smells in VR projects?*” We investigated the distribution of smells to find out whether there is a set of code smells that occur more frequently in VR systems.
- **RQ₃** : “*Can we draw a relationship between code metrics and fault proneness?*” It is commonly believed that code metrics and fault-proneness, i.e., if a set of code metrics reaches a predefined threshold, it is very likely that the project could also have some defects. We investigate this using an unsupervised defect prediction approach.

B. Study Design

The process of selecting open-source projects consists of a systematized search, on *Github*, using the keywords “virtual reality” and “VR”. With the objective of drawing a more specific profile, the search focused only on projects developed for the *Unity* platform, since it has emerged as the most popular VR development platform due to its extensive documentation [11].

Our primary aim is to explore virtual reality projects from both project and source code perspectives. To do so, we cataloged and analyzed a total of 151 open-source projects, available in *Github*. Some of the projects could not be analyzed due to either missing external dependencies or custom-build

mechanisms (i.e., missing standard C# project files), thus we were able to analyze a total of 119 projects.

In order to draw a picture concerning research questions *RQ₂* and *RQ₃*, we also cataloged a set of general (Non-VR) open-source projects, which have similar characteristics (same C# programming language). The goal is to try to compare the information observed in the VR application with Non-VR application. Therefore, we catalog a total of 177 Non-VR projects. After an individual process analysis, we removed duplicated projects and projects that had missing external dependencies or custom-build mechanism. In the end we achieved a total of 107 Non-VR projects able to be used in our experiment.

Since our goal is to analyze which impacts the lack of software testing practice can cause on VR projects, concerning Non-VR projects, we will use only data related to classes that have been properly tested.

C. Overview of Projects

Table I presents general information about the projects analyzed.

Table I: General characteristics of the analyzed projects.

Attributes	VR	Non-VR
Projects	119	107
Number of tested classes	63	4,186
Number of classes (total)	21,508	21,563
Lines of code (C# only)	2,314,522	2,455,766

Aiming to estimate non-functional requirements used to evaluate the performance of a system, such as software quality attributes of the projects, and to give an overall view of the projects analyzed, we computed, according to an object-oriented design [12], a set of metrics that are summarized in Table II.

Table II: Metrics of the analyzed projects.

Metric	VR	Average	Non-VR tested	Average
Number of Children	3,811	0.17	716	0.17
Number of Fields	102,785	4.77	7,062	1.68
Number of Methods	107,516	4.99	14,374	3.43
Number of Properties	24,395	1.13	67	0.01
Number of Public Fields	49,793	2.31	1,368	0.32
Depth of Inheritance Tree	4,072	0.18	1,278	0.30
Number of Public Methods	65,624	3.05	9,582	2.28
Lack of Cohesion of Methods	34,197	1.58	7,958	1.90
Weighted Method per Class	190,658	8.86	21,959	5.24

All the information about the projects, the data used for plotting the tables and graphs, as well for the discussion, are available in the experiment repository³.

IV. RESULTS AND DISCUSSION

In this section, we address the research questions and further discuss the results obtained from the analysis and our observations with the empirical study.

³https://github.com/stevao-andrade/ACL_defect_prediction

A. How is VR software tested?

Regarding the first research question (**RQ₁**) of the study, which aims to understand the question: “*How does testing happen in open-source VR software systems?*”, the 119 projects were manually evaluated and it was found that only 6 VR projects (*Bowlmaster* - 53 tests, *CameraControls* - 60 tests, *GraduationGame* - 15 tests, *MiRepositorio_VRPAD* - 11 tests, *space_concept* - 11 tests, *UnityBenchmarkSamples* - 4 tests) are concerned with the software testing practices, including a total of 154 unit test cases, to evaluate the projects’ functionalities.

Despite the existence of unit testing, we were unable to calculate information regarding testing criteria, such as code coverage, since this Unity does not provide an out-of-the box solution to code coverage.

Based on the information collected, it can be observed that from the 119 analyzed projects, only 6 (5.04%) have some software testing activity, and even the projects that have test cases, do not present many tests that can ensure that the main functionalities of the applications were adequately tested. Based on this observation, concerning *RQ₁* we came to the conclusion that there is not yet consensus regarding the application of software testing practices for VR applications and this motivated us to explore the next research questions.

These results are in agreement with the most recent papers in the literature. Karre et al. [13] conducted an empirical study of VR practitioners to learn the current practices and challenges faced in industry. The software testing related results points out to the absence of adequate tools, as well as uncertainty about how to test the VR app apart from conducting a standard field evaluation. As a consequence, this lack of usability evaluation methods and automated testing tools tend to cost a lot of time to release a VR product.

In order to understand the risks and advantages of these characteristics and to accurately answer *RQ₂* and *RQ₃*, in the next sessions, we compare the difference between the VR projects and Non-VR projects concerning code smells and fault-proneness distribution.

B. Distribution of Code smell

Observing the lack of software testing practice in all the other projects, we decided to investigate how this practice is reflected within the projects. To do so, we decided to measure the incidence of code smells [14] within the projects investigated. This leads us to the second research question (**RQ₂**) presented: “*What are the distribution of architecture, implementation, and design smells in VR projects?*”.

In order to better understand what is related to the lack of tests, we compared the results obtained in the VR projects with the results obtained in Non-VR applications, which have well-defined test cases within the projects.

In general, the presence of code smells in software projects indicates the presence of quality problems. Besides, it also increases the risk of software faults [15]. Such problems directly impact features such as maintainability and contribute to make it difficult for software to evolve.

To better understand **RQ₂**, we identified three different types of code smells in the projects:

- **Architecture smells:** focus on identifying points of interest for possible structural problems that can negatively contribute and hamper activities such as debugging and refactoring, as well as increasing the cost for fault correction and refactoring, due to the characteristic of increasing the complexity of the software, when present [16].
- **Implementation smells:** code smells or implementation smells were first introduced by Fowler et al. [14] and seek to establish a concept to classify shortcomings in object-oriented design principles. This class of smells covers principles such as data abstraction, encapsulation, modularity, hierarchy, etc.
- **Design smells:** are specific types of structures that may indicate a violation of a fundamental principle, which can impact aspects of design quality [17].

In order to calculate the distribution of the code smells previously described within the projects, we use the *Designite* tool [18]. The smells were classified according to the number of occurrences in the analyzed classes and percentage distribution. The data is presented in Tables III, IV and V.

It is worth mentioning that test case classes were not taken into account for this smell classification, once our initial target was to measure the quality aspects of the source code classes. Besides that, smells in software test codes require a whole different classification approach [19].

Table III: Description of the detected architecture smells and their distribution.

ID	Smell	VR	Non-VR tested
AAI	Ambiguous Interface	29	0.13%
ACD	Cyclic Dependency	212	0.99%
ADS	Dense Structure	3	0.01%
AFC	Feature Concentration	366	1.70%
AGC	God Component	201	0.93%
ASF	Scattered Functionality	84	0.39%
AUD	Unstable Dependency	78	0.36%
Std Dev		118.34	7.17
Average		139.0	9.14

It can be observed that in Table III, among the VR projects, there is a low incidence of architecture smells, with only three types (ACD, AFC, and AGC) presenting a percentage of occurrence between 0.93 % and 1.70 %. Observing the Non-VR projects, it can be observed that this category of smells had a lower incidence compared to VR projects. The AUD, AGC and AFC smells showed the highest occurrence rates, with percentages between 0.26% and 0.57%.

VR project behavior can be justified due to the fact that within the *Unity* platform, although an object-oriented language (C#) is mostly used, the development model is considered a component-based programming approach. This approach focuses on the separation of concerns regarding the features to be developed in the system.

Despite Non-VR applications presenting lower rates of architecture smells, it mainly shows a higher incidence of smell AFC. This smell occurs when a component performs more than one architectural concern/feature . This can be

explained due to the programming model adopted. A large part of Non-VR projects corresponds to web applications, which typically use a Model-View-Controller (MVC) standard for application development. As shown by Aniche et al. [20], systems that adopt such architecture can be affected by types of poor practices that lead to the apparition of such a smell.

From a software testing point of view, the lower rate of architecture smells can be considered as a decisive successful factor, since the low dependence between modules is a characteristic that facilitates the application of unit tests [21]. In general, when it is necessary to communicate with other units of code, sometimes *stubs* or *mock objects* are used to represent this communication.

In order to better understand the presented results, regarding each of the classes of smells analyzed, we verified if there is, in fact, a statistical difference between the presence of smells between groups of classes that were not tested and groups of classes that were tested during its development process. Therefore, due to the low number of smell types for each category (architecture, design and implementation), and since we can not guarantee that the data collected departs from a normal distribution, we applied the Mann-Whitney test [22] to verify whether there is a statistical difference between the presence of smells for each category of smells evaluated.

The null hypothesis (H_0) of the Mann-Whitney test indicates that “*The distribution of the variable in question is identical (in the population) in the two groups*”, that is, there is no difference in the presence of smells between classes that have not been tested and classes that have been tested and the alternative hypothesis (H_1) indicates that “*The distributions in the two groups are not the same*”, therefore, there is a statistical difference between the incidence of smells for classes that were not tested against classes that were tested.

Considering the value of alpha = 0.05, which comprises the complement of the margin of a confidence level of 95%, for the architecture smells, H_0 with a p-value = 0.00760 could be rejected. Thus, it indicates that there is a statistical difference between the presence of smells when comparing architecture smells in classes that were not tested against classes that were tested.

Using a descriptive analysis, obtained by analyzing the number of occurrences of each type of smells, it could be observed that classes that were not tested tend to present a higher rate of architecture smells in relation to classes that were tested.

It can be observed that, different from the architecture smells, in Table IV, we can identify a high rate of implementation smells in the VR projects. We highlight ILI, ILS, and IMN, which had a percentage of occurrence of 31.81%, 55.55%, and 117.46%, respectively.

Although it does not pose a direct risk to the source code produced, smell ILI may be an indicator that something can be revised/refactored. A very long identifier may be an indication that there is a need for too much text to distinguish/identify variables and in some instances, this may indicate that the

Table IV: Description of the detected implementation smells and their distribution.

ID	Smell	VR	Non-VR
ICM	Complex Method	1,812	8.42%
ICC	Complex Conditional	684	3.18%
IDC	Duplicate Code	9	0.04%
IECB	Empty Catch Block	150	0.70%
ILM	Long Method	583	2.71%
ILPL	Long Parameter List	2,117	9.84%
ILI	Long Identifier	6,841	31.81%
ILS	Long Statement	11,947	55.55%
IMN	Magic Number	25,264	117.46%
IMD	Missing Default	931	4.33%
IVMCC	Virtual M. C. C.**	35	0.16%
Std Dev		7,425.91	11.87
Average		4,579.36	14.63

**Virtual Method Call from Constructor

programmer may not be using the most suitable data structure to represent it.

ILS occurs when there is an excessively long statement. Long declarations tend to make it difficult to manage the code and are consequently villains if observed from the practice of software testing. Very long code snippets tend to be harder to test because they often become too complex when compared to smaller snippets that are managed more efficiently.

Finally, IMN occurs when an unexplained number is used in an expression. In general, magic numbers are unique values that have some symbolic meaning. Good programming practices indicate that in these cases, such numbers should be declared as constants to facilitate the reading of the source code, as well as to standardize its use.

Non-VR projects again presented a lower occurrence rate. The most frequent smells were ILS, IMN and IMD which achieved, respectively, percentages of 0.96%, 0.85%, and 0.65%.

The appearance of this type of smells is connected with the lack of guidelines for standardization of code as well as the lack of code refactoring practices. From the standpoint of software testing, opting to use of constants instead of magic numbers can ensure that once the value of the constant has been tested, there is no risk that the value of the constant is erroneously declared in the future.

We also applied the Mann-Whitney test to verify whether there is a statistical difference between the presence of implementation smells in groups of classes that were not tested when compared to the classes that were tested. Adopting a confidence interval of 95%, the test presented the p-value = 0.00040, which rejects the null hypothesis of the test and confirms the data presented in Table IV, proving that classes that were tested tend to present a lower rate of implementation smells.

Finally, we have the design smells which seek to identify breaches of design principles. It can be concluded from Table V it is possible to conclude that this class of smells was the one that presented the highest degree of incidence in the VR projects. DUA, DTA and DDE smells were the ones with the highest percentage of occurrence with 17.39%, 32.49%, and 37.67% respectively.

The DUA smell deals with the practice of unnecessary abstractions and is identified when an abstraction has more

Table V: Description of the detected design smells and their distribution.

ID	Smell	VR	Non-VR
DBH	Broken Hierarchy	245	1.14%
DBM	Broken Modularization	991	4.61%
DCM	Cyclically-dependent M.	3,149	14.64%
DCH	Cyclic Hierarchy	6	0.03%
DDH	Deep Hierarchy	0	0.00%
DDE	Deficient Encapsulation	8,101	37.67%
DDA	Duplicate Abstraction	2,469	11.48%
DHM	Hub-like Modularization	4	0.02%
DIA	Imperative Abstraction	627	2.92%
DIM	Insufficient Modularization	1,171	5.44%
DMH	Missing Hierarchy	18	0.08%
DMA	Multifaceted Abstraction	209	0.97%
DMH	Multipath Hierarchy	1	0.00%
DRH	Rebellious Hierarchy	389	1.81%
DUE	Unexploited Encapsulation	15	0.07%
DUH	Unfactored Hierarchy	483	2.25%
DUA	Unnecessary Abstraction	3,741	17.39%
DTA	Unutilized Abstraction	6,987	32.49%
DWH	Wide Hierarchy	64	0.30%
Std Dev		2,344.41	14.59
Average		1,508.94	14.00

than one responsibility attributed to it. This smell tends to occur when there is an application of procedural programming features in the context of object-oriented programming languages [17].

From the standpoint of VR applications that adopt component-based programming, the appearance of this smell can be explained by the fact that the programming approach focuses on creating interchangeable code modules that work almost independently, not requiring that to be familiar with their inner workings in order to use them.

Unnecessary design abstractions increase their complexity needless and affect the comprehensibility of the overall design. From a software testing point of view, this bad practice tends to hamper test practices

DTA occurs when an abstraction is left unused, is not being used directly, or because it is not reachable in the source code. This smell correlates with DUA since unnecessary abstractions tend not to be used. Another impact factor for the appearance of this smell is linked to possible code maintenance/refactoring activities, which tend to leave traces of code that are no longer needed.

From the standpoint of software testing, the existence of a test base that can be used as a regression test tends to facilitate the localization of source code that is no longer necessary, causing the occurrence of this smell to be reduced. From a tester's point of view, if there is a code that is not being used in the project, it does not need to be tested. Therefore, identifying these snippets of code can lead to more efficient testing activities.

Finally, smell DDE, which identifies cases of poor encapsulation, had the highest occurrence rate in this class of smells. This smell occurs when the declaration of attribute visibility of a class is more permissive than necessary. For example, when the attributes of a class are unnecessarily declared as public.

From the standpoint of software testing, separation of interests allows implementation details to be hidden. If an abstraction exposes implementation details unnecessarily, it leads to undesirable coupling in the source code. This will have

an impact on the testing activity because checking units that have a high degree of coupling becomes a more challenging task due to the need for more complex mocks and stubs.. Similarly, the high degree of coupling causes changes that are made in a code snippet to reflect in various parts of the application causing previously designed tests to fail if they are not adequately designed.

Non-VR applications had a lower occurrence in this category of smells, in which DBM and DCM are the two that presented the highest occurrence, with 1.10% and 1.08% respectively. The explanations for the occurrence rate for the DCM smell are related to the cyclic dependence issue of the MVC model and the DBM smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions.

Once again, we applied the Mann-Whitney test to check whether the data obtained from our empirical evaluation can draw a real picture about the behavior of class that does not have tests compared to classes that were properly tested. Once again the Mann-Whitney test proved with a p-value = 0.00089 that classes that were tested tend to present lower rates of smells when compared to classes that were not tested.

Our second research question (*RQ₂*) sought to understand “*What are the distribution of architecture, implementation and design smells in VR projects?*”. We investigated the main types of smells for VR applications and compared their results with Non-VR applications. We also presented a discussion about how software testing practice can benefit from avoiding the smells that obtained the highest occurrence rate.

According to Hall et al. [15], code smells have a significant but small effect on faults. This can justify the fact that Non-VR application projects, which have test cases, present a lower rate of code smells when compared to VR applications, which do not have, for the most part, a well-defined test activity. However, the presence of smells not only hides potential source code flaws but also contributes to hindering the maintainability and evolution of the source code in larger projects. This leads us to the last research question of this study (*RQ₃*), which aims to investigate the fault proneness of VR projects. In a similar way to the analysis of code smells, evaluate this in more depth, we inserted the analysis of Non-VR projects so as to better understand the results.

C. Analyzing fault proneness

As mentioned before, the presence of code smells can indicate the absence of quality attributes in the source code and this can be an indication of faults in a software [15]. Similarly, as previously mentioned, the higher occurrence rate of code smells in the projects can hinder the practice of software testing. To understand the risks of neglecting this activity, we analyzed the projects concerning fault proneness.

Since code smells are identified according to rules and thresholds defined in code metrics [23], we aim to investigate (*RQ₃*) the question: “*Can we draw a relationship between code metrics and fault proneness?*”. To do so, we use the code

metrics described in Table II with a fault prediction technique, which uses the metrics value as an indicator to suggest whether a given source code is fault-prone or not.

By exploring relationships between software metrics and fault proneness, we seek to justify the need for software testing activities. For instance, a high threshold in a specific metric may lead us to suspect, with high probability, about the reliability of some parts of the code.

The effectiveness of fault prediction techniques is often demonstrated using historical repository data [24]. However, once these techniques are adopted, it is not clear how they would affect projects that do not match with the characteristics (language, platform, domain) of the built model [25].

Since we do not have access to a dataset or a bug track history maintained with VR systems data, we tried to exploit an approach that uses an unsupervised fault prediction technique [26], that does not rely on historical data, to investigate fault proneness on the analyzed projects.

We use the Average Clustering and Labeling (ACL) [26] approach to predict fault proneness in unlabeled datasets. ACL models obtain good prediction performance and are comparable to typical supervised learning models in terms of precision and recall, offering a viable choice for fault prediction when we do not have historical data related to faults.

This study can help software developers to understand the characteristics of VR software and the potential implications of neglect software testing activities. Raising awareness is the first step towards VV&T activities.

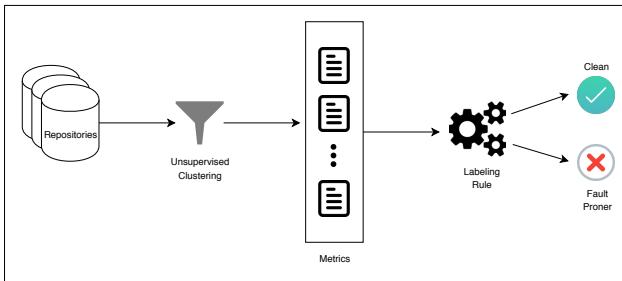


Figure 1: General process of ACL fault prediction approach.

Figure 1 describes the process used by the approach to attest if a given instance of code is defined as fault-prone or not. In general terms, the algorithm of the approach calculates the average value for each of the code metrics used, it builds a violation matrix metric, calculates metrics of instance violation and finally defines whether the analyzed instance is considered as fault-prone or clean. Details of the implementation can be found in [26] and in the repository that contains the information about this work ⁴.

The 119 VR projects were analyzed using the described approach and according to the classification metric adopted, from 21,508 classes contained in all the projects, a total of 2,627 classes or 12.21% were classified as classes with a high

⁴https://github.com/stevao-andrade/ACL_defect_prediction

probability of having faults, due to the fact they extrapolate the threshold defined by the approach to consider them as clean.

Similarly, in the 107 Non-VR projects, out of 21,568 classes, a total of 1,921 were labeled as fault-prone, which corresponds to a percentage of 8.90% of the analyzed classes.

According to previous investigations [27], the Pareto principle also tends to apply to a software faults context. It is believed that 20% of the files in a project are responsible for up to 80% of the faults found. Therefore, it is natural that the results obtained in the fault-proneness analyze follow this trend.

As pointed out by Nam [28], defect prediction approaches play a role as a complementary approach to help identify potential problems in the source code as well as a mechanism to improve it and consequently get rid of productivity bottlenecks and future issues. Thus, the results presented here are not intended to point out the exact number of problems in a software product evaluated, but to strengthen the hypotheses that projects that adopt quality criteria, such as software testing practice, tend to be less predisposed to future issues.

It's also important to note that, since there is no precise information about the test criteria used in Non-VR projects, as well as any information regarding the coverage reached by the designed tests, it is impossible to guarantee that the tests designed for a class are enough to ensure that it is free of any problems. Therefore, it is natural that the percentage of fault-proneness between projects that have not been tested (VR projects) and projects that have test cases (Non-VR projects) is relatively similar.

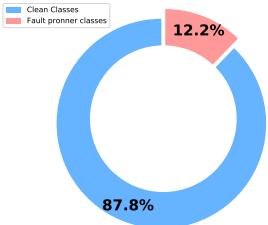
It can be observed that despite having a larger number of classes and lines of code for those analyzed in the VR projects, Non-VR projects presented a lower fault-proneness rate. It is worth noting that the fault-prone algorithm is executed only in the classes related to the source code of the application, thus disregarding the test classes in the Non-VR projects.

This analysis could be an indication that due to the practice of testing, classes of the Non-VR projects have a higher degree of reliability, and therefore are less fault-prone when compared to the classes existing in the VR projects, which mostly do not present test cases.

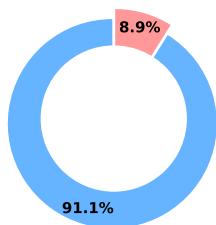
These numbers can be observed in Figure 2a and are alarming numbers since they show the negative impact that a lack of robust and standardized testing technologies can cause to the software industry [29]. Consequently, it contributes to an increase in the incidence of avoidable faults that tend to appear only after the software is used by its end users.

Similarly, the software development cost tends to increase because historically the process of identifying and correcting faults during the software development process represents more than half of the costs incurred during the development cycle [30]. This delay in the product development can lead to situations such as the increase in the time needed to put a product on the market, also resulting in market opportunity losses [31].

We went further trying to understand how the faults pointed out by the approach are distributed into the projects. Since the



(a) VR



(b) Non-VR

Figure 2: Classification of the VR projects according to the ACL approach.

projects have a great variety of sizes, we grouped them into 6 different categories (by the number of classes) in order to observe how the distribution of fault-prone classes occurs.

Figure 3 shows this distribution. It can be observed that in both VR and Non-VR projects, there is a relation between the number of fault-prone classes and the size of the projects. This relation points out that the higher the number of classes in the projects, the higher the average fault-prone classes, and leads us to conclude that neglecting testing activity in larger projects may be even more riskier in terms of the project's success.

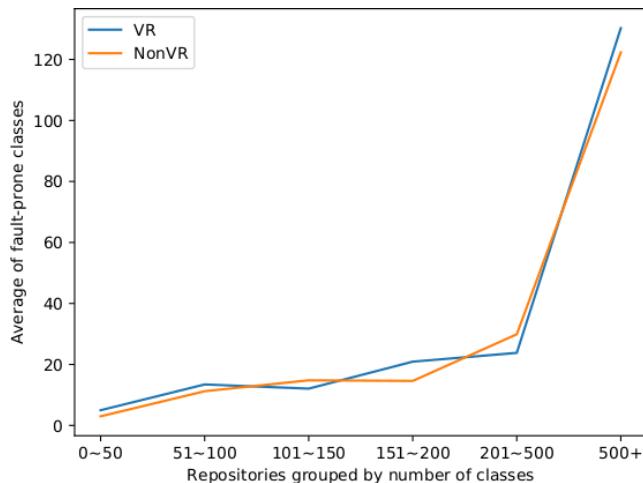


Figure 3: Distribution of fault-prone classes according to the size of the projects.

Future analyses could be extracted from the data obtained. However, we believe that the presented data are capable of attesting a clear answer to *RQ₃*, making it clear that in a general context, the lack of software testing techniques have a direct impact on quality attributes, as demonstrated by the metrics extracted from the analyzed projects and this directly reflects the adoption of bad development practices, which lead to the existence of code smells, consequently becoming an outlet for the increase in faults.

RQ₃ sought to understand the question: “*Can we draw a relationship between code metrics and fault proneness?*” and by implementing the approach to detect fault proneness in the projects investigated, it could be observed that neglecting the

test activity can lead to a higher probability of development problems. According to our analysis, it was seen that the VR projects, which do not present test cases, have a higher propensity to present faults to Non-VR projects, and that propensity tends to increase as the complexity of the projects increases.

It was also observed that although Non-VR projects present test cases in all projects, they still present a high rate of fault proneness. This underscores the importance of the developing software testing practice within the scope of project development. Although Non-VR projects have test cases, the test sets provided do not meet the basic test criteria, such as code coverage, so that part of the code that is not tested is still prone to possible failures.

Another point that this study raised is the need for specific test practices for a specific domain. Software of different domains have different characteristics, which must be adequately investigated. In the context of VR applications, the simple use of unit tests may not be sufficient to attest the quality of the developed product, since the technological advancement has led to the development of systems with advanced features such as images, sounds, videos, and differentiated interaction, presenting new challenges when compared to software testing in conventional domains, such as the lack of information on typical defects and even the lack of a precise definition of a test case and the oracle problem [6], [7].

V. LIMITATIONS AND THREATS TO VALIDITY

The main limitations of this study are related to the fact that the data used in this study were gathered from *Github*. Although the collected data enabled us to discuss the state of practice regarding the application of software testing techniques in the context of VR, open source projects represent only a portion of what is produced in the context of VR applications. Commercial projects and closed projects are also part of this universe, and it is not possible to attest that the results discussed from data extracted from open source projects can be generalized for these other scenarios.

In addition to the limitations described above, another obstacle that must be pointed out is the fact that despite the fact that the assumptions made during the study were related to the context of VR applications, it should be emphasized that all the samples observed only use a single technology (*Unity*), therefore the results indicated here cannot be generalized for other platforms. To achieve this generalization, new studies should be carried out to corroborate or counter the results presented in this study.

Concerning the threats to the validity of this study, which are related both to the evaluation of code smells and to fault-proneness detection, we can highlight the fact of performing an analysis using samples extracted only from a platform and just for open source projects was a significant threat to validity - this relies on the lack of representativity of the projects in serving as a real sample of the universe of all types of projects for the VR domain. Unfortunately, this is a problem that affects

the entire software engineering area, since there is no well-fledged theory capable of ensuring that a set of programs is considered a representative sample for experimentation. To try to mitigate this threat, the most significant possible number of projects was assembled, varying in size (small, medium and large) and application purposes (entertainment, simulation, training, health).

Another measure taken to try to mitigate this threat was to analyze Non-VR projects, which served as subsidies to compare with the results obtained from VR projects, ensuring a better grounded discussion and a minimum baseline for comparison, since, unfortunately, there are still no projects cataloged with VR applications that meet the requirements to be used in this work.

Related to threats to construct validity, possible mistakes can be pointed out both in the analysis of codes smells, as well as in the evaluation of fault proneness. To minimize this threat, the tool *Designite* was used to detect code smells, *Designite* is a commercial tool and has already been successfully used in other experiments [32]. Regarding the approach to detect fault proneness, the strategy was previously validated through experiments in large datasets to attest its efficacy [26], and it is worth mentioning the fact that the main point of the approach is not, in fact, to find faults in the projects, but point out classes that have a high probability of having faults, serving as a guide to direct testing efforts.

Finally, we discuss threats to the internal validity of the study, which are related to the level of confidence between the expected results and the results obtained. The whole study was conducted in a way that minimized this threat. To increase confidence about the presented results, the data were analyzed using tables and graphs and were also made available in a repository to enable the replication if it is deemed necessary.

VI. RELATED WORK

Although virtual reality studies date back a long time [33], only recently have few studies addressed the development of VR applications from a software engineering perspective. The increase in community interest has emerged with the recent popularization of tools that have facilitated access, and consequently developers' interest in this technology, which is still considered as emerging.

Rodriguez and Wang [34] present a survey about projects developed for the *Unity* platform, highlighting the growth of the number of projects in recent years. Another highlight is the fact that despite the higher number of applications focused on games and entertainment, there has been an increase in the number of applications for other purposes, such as training and simulations.

Unlike our work, the paper does not analyze the content of the cataloged material in detail, and is limited to studying the growing trends, development involvement, favorite topics, and frequent file changes in these projects.

Ghrairi et al. [11] conducted a study on the exploratory analysis of *Github* projects and questions extracted from *StackOverflow*, analyzing it from a software engineering point

of view. The study demonstrates the current state of practice regarding the development of open-source VR applications, highlighting mainly the most used platforms and technologies. Moreover, the paper also discusses topics of interest for VR developers by analyzing the VR questions extracted from *StackOverflow*.

The main results show the greater popularity of the *Unity* and *Unreal Engine* platforms as being the most popular among VR application developers. However, they also point out that more work needs to be done to better understand the VR requirements under a software engineering context, which is one of the points that our work seeks to elucidate.

From the perspective of software testing applications in the context of VR, we highlight the work produced by Corrêa et al. [2], which presents a proposal for application software testing for VR applications. This study generates test data using specified requirements through a semi-formal language for VR application development. This approach moves in the direction pointed out by our study, which is the proposition of mechanisms that allow the systematization of the test activity for VR applications.

VII. CONCLUSION AND FUTURE WORK

This paper discusses the main challenges related to using software testing practice in the VR domain. Some of the critical issues related to the quality of these systems were pointed out and possible solutions were also discussed that could be used and adapted to deal with such issues.

We discussed whether there is a real need to test VR systems. To better understand this, a comprehensive study was conducted, guided by 3 research questions, whose objective was: to understand the state of the practice of software testing in the context of VR programs (*RQ₁*), to measure metrics and quality attributes in VR software (*RQ₂*), and finally to evaluate fault proneness in the collection of the software analyzed (*RQ₃*).

In order to answer the raised questions, a collection of 119 VR projects, available in open source projects and manually analyzed, was cataloged to understand the state of the practice concerning the application of software testing techniques. Regarding the application of software testing techniques (*RQ₁*), it was observed that out of all the projects, only 6 of them had some test cases in their project.

Given the results pointed out by *RQ₁*, we decided to evaluate how the negligence of the practice of software testing can be detrimental to a software project, and it was decided to evaluate the distribution of code smells among the analyzed projects. Smells related to architecture, design, and implementation were analyzed. It can be concluded that there is a high incidence of smells in the projects analyzed, especially regarding implementation smells. We discussed the most common smells for each of the categories and how they can discourage the practice of software testing, and also how they can be avoided if a software testing activity is appropriately conducted.

Finally, considering the results of RQ_2 , it was decided to investigate how the lack of good practices and the presence of code smells can impact the quality of the source code produced. To do so, an approach that evaluates code metrics was used to point out classes that are fault-prone (RQ_3). The study pointed out that about 12% of the analyzed VR classes have such characteristics, revealing a significant risk to the success of the projects. The distribution of these classes was also evaluated when observed concerning the size of the projects analyzed. It was observed that the larger a project becomes, there is a higher incidence of fault-prone classes, which may be an indication that neglecting test practices in larger projects becomes even more riskier.

We believe that the results reported in this paper will contribute to raising the awareness of the software testing and virtual reality community about the needs of software testing approaches for VR developers.

Future work intends to propose a fault taxonomy to the context of VR programs. It is believed that having such taxonomy, it would be possible to encourage the development of specific software testing techniques and criteria to the context of VR programs, thus spreading the practice of software testing in order to mitigate possible problems and move towards software projects that best meet software quality requirements.

ACKNOWLEDGEMENTS

Stevão A. Andrade research was funded by FAPESP (São Paulo Research Foundation), process number 2017/19492-1.

We also would like to thanks Tushar Sharma and *Designite* team by providing us an Academic license of *Designite* tool.

REFERENCES

- [1] A. C. S. Corrêa, M. E. Delamaro, and F. L. S. Nunes, "The relationship between requirements engineering and virtual reality systems: A systematic literature review," in *2013 XV Symposium on Virtual and Augmented Reality*, 2013.
- [2] A. C. S. Corrêa, F. L. S. Nunes, and M. E. Delamaro, "An automated functional testing approach for virtual reality applications," *Software Testing, Verification and Reliability*, 2018.
- [3] T. M. Takala *et al.*, "A toolkit for virtual reality software development-investigating challenges, developers, and users," Ph.D. dissertation, Aalto University, 2017.
- [4] Q. Zhao, "A survey on virtual reality," *Science in China Series F: Information Sciences*, 2009.
- [5] A. Bertolino, "Software testing research and practice," in *Abstract State Machines 2003*, E. Börger, A. Gargantini, and E. Riccobene, Eds., 2003.
- [6] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Softw. Eng.*, 1985.
- [7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, 2015.
- [8] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013.
- [9] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014.
- [10] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, 2012.
- [11] N. Ghairi, S. Kpodjedo, A. Barrak, F. Petrillo, and F. Khomh, "The state of practice on virtual reality (vr) applications: An exploratory study on github and stack overflow," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018.
- [12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, 1994.
- [13] S. A. Karre, N. Mathur, and Y. R. Reddy, "Is virtual reality product development different?: An empirical study on vr product development practices," in *Proceedings of the 12th Innovations on Software Engineering Conference*, 2019.
- [14] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.
- [15] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, 2014.
- [16] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, 2015.
- [17] G. Suryanarayana, G. Samarthym, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.
- [18] T. Sharma, P. Mishra, and R. Tiwari, "Designite - a software design quality assessment tool," in *2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE)*, 2016.
- [19] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.
- [20] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, and A. van Deursen, "Code smells for model-view-controller architectures," *Empirical Software Engineering*, 2018.
- [21] M. F. Aniche, G. A. Oliva, and M. A. Gerosa, "What do the asserts in a unit test tell us about code quality? a study on open source and industrial projects," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013.
- [22] M. P. Fay and M. A. Proschan, "Wilcoxon-mann-whitney or t-test? on assumptions for hypothesis tests and multiple interpretations of decision rules," *Statistics surveys*, 2010.
- [23] F. Khomh, M. D. Penta, and Y. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *2009 16th Working Conference on Reverse Engineering*, 2009.
- [24] S. Herbold, "A systematic mapping study on cross-project defect prediction," *arXiv preprint arXiv:1705.06429*, 2017.
- [25] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Information and Software Technology*, 2015.
- [26] J. Yang and H. Qian, "Defect prediction on unlabeled datasets by using unsupervised clustering," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2016.
- [27] N. Walkinshaw and L. Minku, "Are 20% of files responsible for 80% of defects?" in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018.
- [28] J. Nam, "Survey on software defect prediction," *Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Tech. Rep.*, 2014.
- [29] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, 2002.
- [30] F. Brooks, *The Mythical Man-Month, Anniversary Edition: Essays On Software Engineering*. Pearson Education, 1995.
- [31] P. Afonso, M. Nunes, A. Paisana, and A. Braga, "The influence of time-to-market and target costing in the new product development success," *International Journal of Production Economics*, 2008.
- [32] T. Sharma, M. Fragkoulis, and D. Spinellis, "House of cards: Code smells in open-source c# repositories," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017.
- [33] D. K. Boman, "International survey: virtual-environment research," *Computer*, 1995.
- [34] I. Rodriguez and X. Wang, "An empirical study of open source virtual reality software projects," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017.