A Comprehensive Approach to Quality Assurance in a Mobile Game Project

Maxim Mozgovoy
University of Aizu
Aizu-Wakamatsu, Fukushima, Japan
mozgovoy@u-aizu.ac.jp

Evgeny Pyshkin
University of Aizu
Aizu-Wakamatsu, Fukushima, Japan
pyshe@u-aizu.ac.jp

Abstract

Quality assurance is an integral part of software development process. Game projects possess own distinctive traits that affect all stages of work. In this paper, we share the lessons learned during a three year-long mobile game development project. We discuss the conceptual architecture for mobile game quality assurance through the perspective of techniques that turned out to be most efficient for us, including manual testing, automated and manual runtime bug reporting, Crashlytics-supported crash analysis, automated smoke testing, and playtesting. We analyze how these activities address typical game-specific mobile development and testing issues, and why they can be recommended for game projects, as well as for wider range of mobile applications.

CCS Concepts • **Software and its engineering** → **Empirical software validation**; *Software usability*; • **Human-centered computing** → Mobile computing;

Keywords Software development, quality assurance, continuous integration, automated testing, usability, mobile game

ACM Reference format:
Maxim Mozgovoy and Evgeny Pyshkin. 2018. A Comprehensive Approach to Quality Assurance in a Mobile Game Project. In *Proceedings of Central and Eastern European Software Engineering Conference Russia, Moscow, Russian Federation, October 12–13, 2018 (CEE-SECR '18),* 9 pages.
https://doi.org/xxxxxx

1 Introduction

Quality assurance (QA) is a complex set of methods, used in all stages of software development, ranging from requirements engineering and software design to coding and testing. Quality assurance practices cannot guarantee perfection of a software product, but they naturally affect its quality and reliability. Explicit quality assurance measures are found in all widely used software development processes, from traditional waterfall model to modern agile approaches where testing procedures are moved to earlier stages of software lifecycle [10, 12].

Still, quality issues are common in resulting software products. Khalid et al. [13] analyzed user reviews of 20 most popular iOS apps of June 2012. They found that 26.68% of user complaints are related to functional errors, and other 10.51% of complaints mention app crashing. Together with "feature request", they constitute top 3 complaint types.

One may argue that the best way to ensure software quality is to maintain high standards of software development culture [15]. Indeed, poor design and planning, and somewhat relaxed attitude to writing code is often mentioned as the primary reasons for buggy software [17]. Thus, gradual improvement of software development processes is a necessary, but difficult and time-consuming measure.

Many QA-related issues are common for all types of software. However, mobile application testing has its own challenges. Research on advancing low-cost sharable open source solutions that enable scalable process of mobile application testing is one of clear trends in mobile software industry, where, due to some limitations on applicability of automated testing tools for mobile applications and lack of comprehensive solutions for mobile testing, many teams still prefer to rely on manual testing practices [16, 21].

The analysis of numerous projects dedicated to mobile software testing helps us to discover certain particular aspects of mobile testing, attracting attention in recent works. Since there is a large number of mobile devices with a diverse specifications, developers have to be sure that the application works properly on different real devices, and the emulators are often not enough for this purpose.That is why one of major concerns is how to support running automated tests on multiple devices in case of resource-intensive mobile applications [14]. The difficulties of deployment of a continuous integration framework meet other specific problems of mobile testing, including power consumption issues. Application crashes is another serious issue, important for mobile applications, since the developers have to support a huge variety of devices with their own quirks. Thus, crash reporting components, which would provide the detailed description

CEE-SECR '18, October 12–13, 2018, Moscow, Russian Federation

ACM ISBN xxxxxx…$15.00
https://doi.org/xxxxxx

of the failure context, are crucial for a continuous integration pipeline [20]. Crash reporting is a nontrivial problem, since the reports might require inclusion of a lot data, such as screenshots, steps to reproduce the failure, exception stack traces, etc.

In turn, mobile games demonstrate many important particularities and challenges of the testing process, such as:

- organization of UI and usability testing, including playtesting;
- automated non-native GUI testing relying on image recognition algorithms to support simulation of user interaction with hand-drawn elements of the game;
- organization of parallel testing on multiple devices, possibly including both real and emulated hardware;
- existence of resource-intensive tests, such as stress tests and/or tests requiring intensive data exchange;
- maintenance of device health and acceptable battery levels of test devices.

These particularities might not explain all the most important mobile testing issues, but they are surely connected to the significant factors of entertainment software testing.

Linares-Vásquez et al. [16] suggested a conceptual architecture for mobile testing that follows a number of principles forming the triad "continuous – evolutionary – large-scale". The first term ("continuous") connects the problem to continuous integration methods: applications are continuously tested under changing environmental conditions with modeling realistic usage scenarios and triggering a testing iteration according to the current version of the application and/or different usage conditions (operating systems, devices, etc.). The second term ("evolutionary") refers to automatic adaption of testing tools to the changes in software: *"app source code and testing artifacts should not evolve independently of one another"* [16]. Finally, the "large-scale" principle refers to a large number of issues related to continuous delivery, including the support for parallel testing on multiple devices, both physical (preferred) and virtual, performance and stress testing.

We concentrate on relatively simple, but cost-efficient measures, aimed to reveal bugs before they creep into the release version, and to facilitate quick fixes of bugs not identified during testing. While all these methods are well-known, they deserve additional discussion within the process of game development, since it has certain distinctive traits that affected our views on quality assurance. We believe that these observations will be helpful not only to game programmers, but also to the developers of general-purpose interactive desktop and mobile software with frequent releases and high requirements for usability and quality.

Our observations were made during the development of a mobile game *World of Tennis: Roaring '20s* created by a small team in three years. The game is written in Unity game engine [30], and is currently available for iOS, Android, and Universal Windows platforms. The game is free to play with additional in-app purchases. It requires Internet connection for most actions, though occasional lags and unstable connectivity do not affect game process. One of the most interesting aspects of this game is a machine learning-based AI system that observes players' behavior to substitute them in player-vs-player matches [23]. This capability allows the players to compete against each other at any time, and mitigates all negative effects of poor Internet connection. This feature also requires implementing particular quality assurance scenarios, including believability tests in addition to more commonly used playtesting or GUI testing.

2 Mobile Software QA Process for Game Developers: A Conceptual Architecture

The nature of a software product we create affects the whole development process, including quality assurance. As Ben-Ari notes discussing the failure of Ariane 5 rocket launch in 1996, *"the bug [was] not caught during testing... [because] you cannot debug the system by inserting breakpoints while the rocket is being launched!"* [5]. Game development has its own peculiarities, discussed in literature [25, 28]. With respect to these peculiarities, we describe a conceptual architecture for a solution aimed to combine necessary practices of mobile software QA (see Figure 1).

This solution includes a number of several (not completely independent) views: change management, test management, continuous integration core, device management, and mobile device testing issues.

Monitoring changes section describes activities and processes that should trigger testing operations. Such processes are connected to different change sources, including the project source code, game design artifacts (e. g. the game UI), target platforms and environments, and third-party libraries and tools. Changes in software and in testing artifacts are also triggered by collecting user feedbacks received after playtesting sessions.

Managing tests section encompasses processes monitored for changes, activities on test script creation or modification aimed at arranging different types of required tests, including automated unit and smoke tests, usability tests, playtests and manual tests. The test scripts are to be stored in a test artifacts repository and run on devices under testing.

Continuous integration core section describes the common architecture for a CI solution supporting automated testing process. This solution is based on established practices used by many software teams and includes a number of mandatory components of CI pipeline, such as build machine (ensuring the generation of fresh builds triggered by the changes in source code), test runner (executing test scripts and generating test reports), test servers hosting mobile test automation frameworks, and connected devices where the game under test is being installed and run.

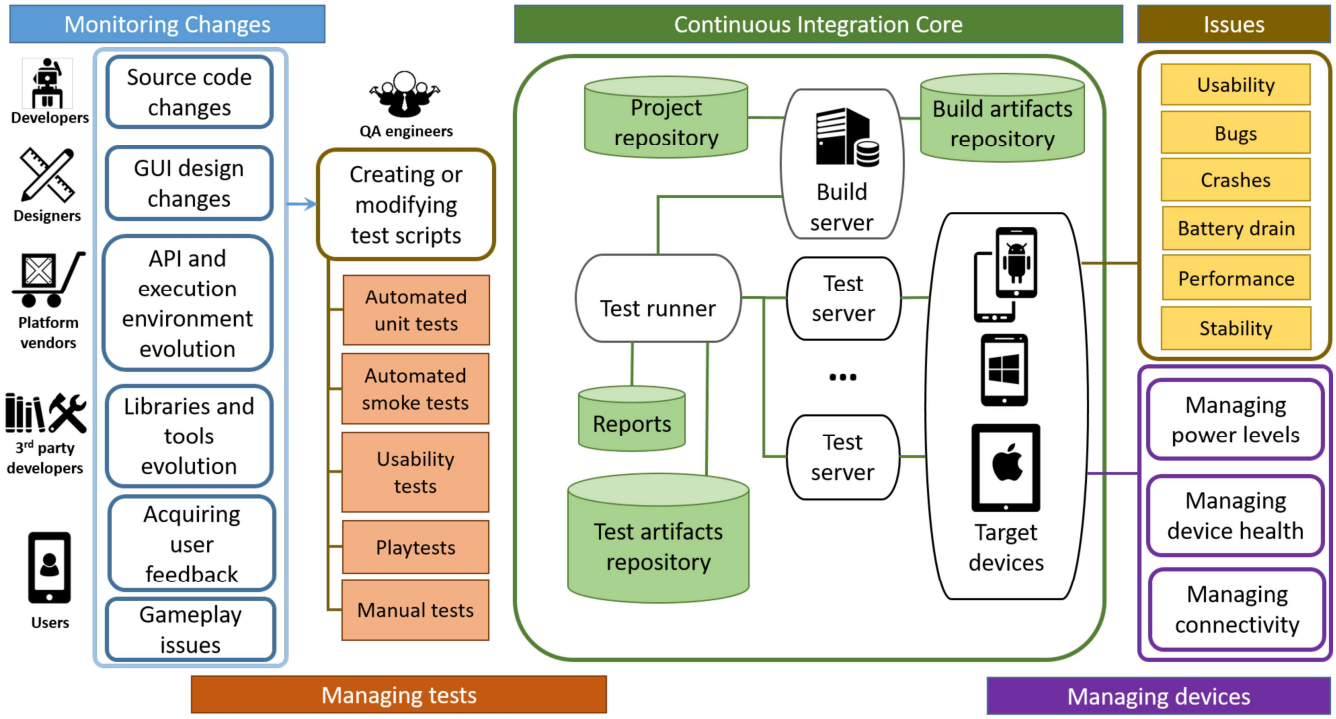

Figure 1. Conceptual architecture for mobile software QA process.

Managing devices and *Issues* sections describe the concerns that should be taken into account when dealing with real mobile devices as target test vehicles.

Hereafter we list the most significant factors affecting our approach to quality assurance. We argue that these factors can explain a number of important recently recognized specific aspects of mobile game software testing.

2.1 Heavy reliance on unstable 3rd-party libraries and tools

We are responsible for most errors in the game, but 3rd-party tools are not perfect, either. We are forced to use certain libraries to integrate with external services (such as Google, Facebook or ad providers), and we have to rely on Unity capabilities for internal game engine functionality and cross-platform development. Some of these modules are quite complex (such as cloth simulation), immature, and sometimes cause app crashes. Often we have to face a choice: either to use a 3rd-party module that implements a functionality needed for a certain feature, or to cut this feature at all. It is also often means to rely on early unstable versions of 3rd-party modules and to hope that they will be improved in the future. In practice it means that our approach to functional errors and crashes has to be nuanced. For example, we might decide to tolerate a certain level of crashes if it lets us to integrate with an ad provider or enable great-looking cloth simulation.

2.2 Diversity of hardware and software platforms

Unity does not make cross-platform development *trivial*, but it certainly makes it *much simpler*. Therefore, it is very tempting to take advantage of this capability, and to release the game on a wide range of platforms. In turn, it means that the game has to be tested on each platform separately. Platform-specific errors typically occur in fragments of code appearing in libraries compiled as native binaries and in procedures working with platform-specific SDKs (e. g, for in-app purchases). Diversity of hardware and operating systems also imposes additional challenges. Some distribution channels such as Apple and Google application stores allows the developers to specify the types of compatible devices by providing minimal supported screen dimensions, amount of RAM available, target operating system versions, and so on. It leads us again to treat known flaws pragmatically. If the game does not work properly on certain types of devices, it might be reasonable from a business perspective to consider them incompatible rather than invest efforts into patches.

2.3 Abundance of visual and sound issues

A great number of bugs in games can only be revealed with manual testing. For instance, we had situations when shadows were not visible, the colors of clothes were wrong, the characters had their feet below the ground level, some text boxes overlapped with other GUI elements or were too small to contain the corresponding text lines. Same can be said

about animation: certain sequences looked unnatural and transitions between animation types were not smooth. Sound effects also were sometimes missing or different from what was expected. Therefore, automated testing in game projects is applicable to a relatively narrow set of cases. Ironically, this factor motivated us to automate as many scenarios as we could to give our testing team more time and incentive to find nontrivial bugs.

2.4 Large proportion of high-cost unit testing code

Unit testing is one of the most popular quality assurance instruments, associated with agile software development methodologies [8]. However, as observed by Sanderson, different types of code have different costs and benefits of being covered by unit tests [29]. Sanderson identifies two types of code with high cost of unit testing: complex code with many dependencies, and trivial code with many dependencies ("coordinators" between other code units). According to Sanderson, complex code with many dependencies should be refactored to separate algorithms from coordination.

Our experience shows that a game project has a large proportion of both types of high-cost unit testing code. We believe the primary reason for it is that the most cost-efficient type of code ("complex code with few dependencies" in Sanderson's scheme) belongs to the game engine such as Unity and 3rd-party libraries. The problem is further aggravated with the fact that "complex code with many dependencies" is rarely refactored in practice and thus also cannot be unit-tested efficiently.

It might be tempting to attribute the lack of refactoring and frequently noted substandard design of system architecture in game projects to low culture of development. However, there are objective factors contributing to this situation. In particular, games have to be *entertaining* and provide *excitement* – requirements that can hardly be satisfied with traditional planning methods. Therefore, game programming requires a lot of experimenting, and it is not surprising that the developers tend to view much of their work as "throw-away code". As one of the respondents of the study [25] puts it, *"[the producer] doesn't want you [the developer] to go and spend a whole bunch of time planning out how you're going to do this thing that he's asked for because he might change his mind in a week or two. Knowing that, he knows deep down that designing is useless because he's going to be constantly changing things"*. Furthermore, there is little incentive to refactor working code since most of it is project-specific and most probably will not be reused: *"there's always a feeling in games that you almost don't really have to maintain it... there's kind of a sense that you're the last one to touch the code"* [25].

2.5 Deep integration of GUI and animation

Automated tests (especially unit tests) often rely on the possibility to separate entities. One might want to test game physics separately from animation or GUI independently from underlying logic. However, it might be virtually impossible to do in a game. For instance, in Unity animation is an integral part of character motion model. To check the changes in character's coordinates during movement, one has to play the related animation sequence. The notion of "user interface" is also vague in games, as any clickable on-screen object can be considered a part of interface. Furthermore, typical user controls like buttons or edit boxes are often hand-drawn in games and thus inaccessible through standard automation interfaces (such as UI Automator in Android or XCTest in iOS).

3 QA in *World of Tennis*

Following the principles demonstrated in the conceptual architecture, we implemented a practical solution for mobile game QA, with combining a number of approaches and tools fitting well our project goals (Figure 2).

In this section, we will discuss in more detail some specific measures we implemented in the project. We consider them useful and cost-efficient, and are willing to adhere to the same practices in the future.

3.1 Crashlytics Crash Reporting

As mentioned in the previous section, we take a pragmatic approach to errors. With numerous 3rd-party modules we use, Unity as a game engine, and a variety of supported platforms and devices, malfunctions are inevitable. Our task from the early stages of development was not only to identify faults, but also to assess their severity for the product.

One of our first decisions was to integrate Crashlytics crash reporting service [2]. It embeds special crash reporting code into the application, which sends crash details into a central server. As developers, we can analyze the reasons of crashes and the list of devices where crashes occur.

In particular, Crashlytics helped us to identify devices having not enough RAM for our game. On mobile platforms, a task scheduler can simply kill a foreground application if it consumes a critical amount of memory, which is is practically equivalent to a crash. However, it is not easy to decide where exactly one has to draw a line, s ince numerous devices belong to a "gray area" where crashes are possible, but not certain. Actual statistics from Crashlytics helped us to make a well-grounded decision.

3.2 Autobugs and Manual Bugs

Developers widely use *assertions* to check certain assumptions about certain points in code. Assertions can be seen as a part of "design by contract" approach, introduced by Meyer [18]. While there is a general agreement that assertions should be used during development as a method for both in-code documentation and quality assurance, there is a debate on whether assertions should be preserved in production code [7]. The arguments in many cases depend on what

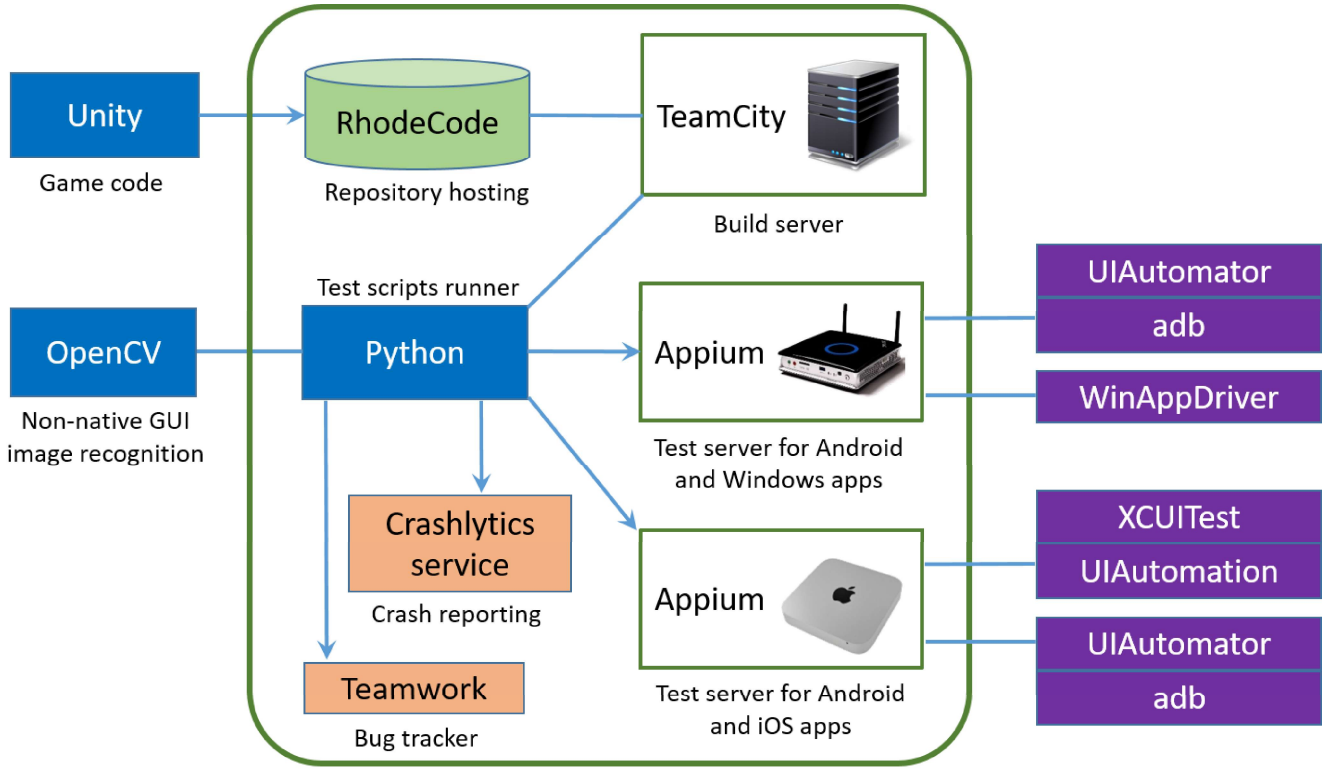

Figure 2. Practical solution for mobile game QA process.

assertions *actually do*, and the typical presumption is that a failed assertion shows an error message and terminates the application.

We decided that each failed assertion and each raised exception should be reported to us. Fortunately, we presume the presence of Internet connection on user devices, thus error reporting is easy to automate. We use Teamwork [3] as a task and bug tracker. It has a useful capability to create tasks via email messages: an email with a predefined structure sent to the `tasks.teamwork.com` domain becomes a new task. This way, we created a dedicated "Autobug" category of tasks to gather information about failed assertions and raised exceptions. Each report contains basic information about the build, user device, and current user account. It also contains a link to the detailed session report stored on our dedicated server.

The same technology is used for the "Manual Bug" category. The users marked as beta-testers in the system have an option to pause the game at any moment and send a bug report. It will be posted to Teamwork in the same manner along with the session report and with a user-supplied description. As noted above, massive manual testing in games is inevitable, so we tried to attract beta-testers as soon as the game became playable. The public announcement of open beta-testing was made one year before the global launch of the game.

3.3 Manual Testing

Our approach to manual testing is relatively straightforward. As soon as we get a new build that is considered "stable", we ask our testers to play several game sessions, noting any problems they encounter. All game sessions are recorded as video clips, and the testers illustrate their findings with links to particular video fragments. Since our QA team is small (only two people test regularly), we also rely on a professional QA company to check our major release builds on a variety of devices and platforms.

3.4 Automated Smoke Testing

Smoke testing is a type of functional testing aimed to reveal failures in a complete system by covering a broad product features with simple test scenarios [11]. A smoke test can be as simple as *"launching the application and checking to make sure that the main screen comes up with the expected content"* [9], it can also evolve into a complex suite of tests checking core application functionality. Microsoft calls smoke testing *"the most cost effective method for identifying and fixing defects in software"* after code reviews [19].

We believe that smoke testing is indeed one of the most cost-efficient methods of quality assurance. In our game, there are certain routine actions that can be automated relatively easily: 1) create a new user and pass the tutorial; 2) play a league match against the next opponent; 3) upgrade

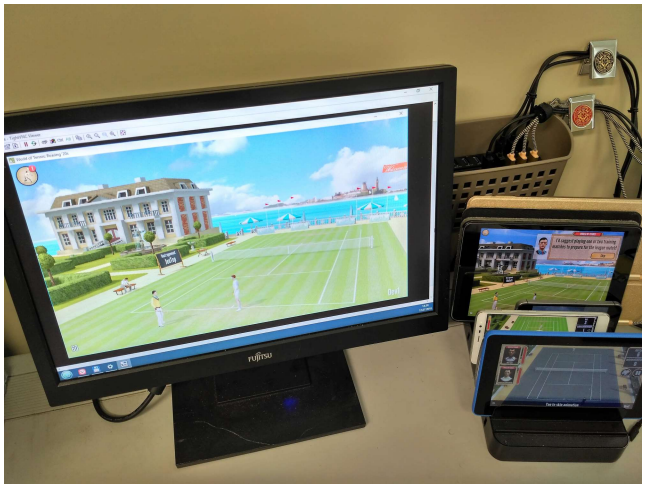

Figure 3. Mobile testing farm.

Figure 4. Usability testing.

your character's skills using available experience points; 4) link your Facebook account to the game; 5) change current club / character / clothes / equipment.

These actions require most subsystems of the game to operate correctly, so it can be expected that such automated testing would identify many critical bugs. Furthermore, automation allowed us to remove the specified scenarios from manual testing, freeing more time of our QA team for more creative bug-finding work.

Technically, mobile smoke tests can be set up using an external service, such as Bitbar Testing [6] or AWS Device Farm [1]. However, we found them too expensive for daily use, and set up our own mobile farm of one Windows, three iOS, and four Android devices (see Figure 3) [24]. The testing farm is fully integrated into our pipeline. When a new build is available on the build machine, the system runs predefined test scripts on all devices in the farm. In Figure 3 we show a number of mobile devices where the mobile game tests run, as well as a desktop version that can be tested by using the same testing farm infrastructure.

The scripts interact with our mobile devices via Appium framework [31] and use image recognition to identify clickable GUI elements. Test logs are available as HTML reports with screenshots, illustrating ongoing actions. If a certain test fails, it is easy to identify the cause in most cases.

It should be noted that these automated tests also generate autobugs, so even if there are no obvious faults reported by the test, it still might detect some malfunction via the mechanism of assertions and exceptions.

We think this system is technologically the most advanced testing mechanism in our daily use. It certainly has numerous flaws: mobile devices require maintenance, and each device might need to be configured in a specific way; Appium and operating system updates sometimes break compatibility, and game updates break existing tests. However, we still consider this solution to be very cost-effective, and are planning to increase the number of implemented tests.

3.5 Playtesting and Usability Testing

All the techniques described above were dedicated to ensure software quality in *objective* terms, such as the proportion of crash-free sessions, the number of known bugs and the probability to face a certain bug, acceptable CPU / GPU / energy consumption levels and so on. However, they cannot reveal *subjective user experience*, which is a crucial topic for games, since they represent leisure activity and thus must be enjoyable. Amaya et al. [4] suggest to consider the separate processes of *usability testing* and *playtesting* as methods to obtain *behavioral* and *attitudinal* data of the users.

The purpose of usability testing is *"to reveal areas of the game in which the player experience does not match with the design intent"* [4]. In other words, the goal of usability testing is to make sure that user interaction with the game is smooth: the users are able to learn game rules, navigate GUI elements to find the functions they are interested in with ease, and in general concentrate on the game process rather than exploring menus and mastering counter-intuitive controls.

In contrast, playtesting is focused on *"players' opinions to illuminate areas of the game in which player experience does not map onto design intent"* [4]. This way, the goal of playtesting is to seek explicit player opinion about their satisfaction with the game and willingness to play.

Briefly speaking, a typical usability testing session consists of recording the users actually playing the game with the subsequent analysis of their actions (see Figure 4). We record the users with two cameras: one facing the user, and another one directly facing a tablet used to play the game. The users are strongly encouraged to verbalize their feelings during the game process. This technique, known as *thinking aloud*, is considered a simple and cost-effective method of revealing cognitive activities of the users [26]. These verbal reports

along with video recordings help us to identify common problems in user interaction with the game.

Playtesting is typically done with structured survey analysis [4]. The participants are asked to play the game for a certain amount of time, and to answer a set of questions, highlighting particular game-dependent concerns. For example, the work [27] discusses a case study of a multi-mission game, where the users had to report their impressions of the "fun", the "excitement" and the "clarity" of each mission's objectives.

In case of *World of Tennis: Roaring '20s*, we had concerns about game tutorial, speed, controls, and AI system, so our playtesting questions were mainly focused on these topics.

Over the course of game development, we have performed three such usability/playtesting sessions, each involving 8–10 participants. Large game development studios suggest recruiting a group of 25–35 people as a pragmatic trade-off between confidence and resource constraints [27]. The results of these tests indeed motivated us to rethink several design decisions, related to the user interface, control scheme, and game tutorial.

4 Future Plan: Health Indicators

Automated smoke testing in practice opens a whole new set of possibilities for more fine-grained types of testing. For example, we keep short summary of all game sessions that include basic data about user device, matches played, their scores and durations, and so on. In particular, such summary includes an average framerate of the game, which is a vital indicator of playability on a particular device. Smoke testing allows us to obtain average framerate values for all test devices for each new build automatically.

This small benefit turned out to be indispensable during certain phases of work. Framerate might drop significantly due to seemingly small scene changes made by the artist or due to different handling of shadows or antialiasing. Furthermore, different devices can be affected differently by these changes. Since our testers possess devices that we consider "typical", they do not always notice framerate drops, as the power of their hardware is still sufficient to run the game smoothly. Thus, it was our deliberate decision to include several low-end devices into mobile farm to make sure the game exhibits acceptable performance there.

We are still experimenting with the game, fine-tuning certain parameters to adjust user experience. Sometimes these adjustments cause unexpected consequences, e. g., significant change of an average match duration (which might not be desirable). Automated smoke test scripts are capable to repeat the same game sessions almost verbatim and thus can provide reliable indicators for average framerate, match duration, percentage of crashes or other faults, and so on. We are planning to introduce a number of "health indicators", and use automated tests to monitor how they change as new game builds are released.

5 Discussion

Mobile free-to-play games is a very special kind of product. They are not designed to be sold once. Instead, their developers have to keep an eye on their audience, tinker with game mechanics and monetization techniques, and implement new features. We believe that such games combine certain features found in both worlds of "games" and "non-game apps".

Research shows that game programmers believe there are substantial differences in their work practice comparing to work practice of people developing office and other non-game applications, and there is evidence that many of these feelings reflect reality [25]. In particular, game projects suffer from loosely formulated requirements, frequent changes of core system elements, heavy reliance on manual testing, and little incentive to improve architecture, since much of the work is seen as "throwaway code". In a sense, a game is like a movie: once it is ready, nobody needs props anymore.

Mobile free-to-play games is not an exception in regards to coding practice, but they at least require strict and reliable quality assurance process to make sure regular updates do not break the game. It is incredibly difficult to establish a place in a hyper-competitive environment of modern mobile app stores, and bugs may cause a quick descent. In addition to negative user reviews, a game might get downranked by a store. In 2017, Google announced that *"higher-quality apps"* will be ranked higher than *"the similar apps that are lower-quality (e.g. if they exhibit more frequent crashes)"* [22].

Therefore, we believe that games would benefit from a more comprehensive approach to testing that takes into account specific issues related to game development. Not all commonly recommended practices are well suitable for game developers, and the right answer to this challenge would be to identify the practices that work best.

6 Conclusions

In this paper, we outlined several objective factors that have a negative effect on mobile game projects. However, they cannot serve as an excuse for functional errors and crashes, haunting many games. Instead, they should be seen as challenges for more comprehensive and streamlined quality assurance procedures, based on cost-efficient measures that take into account the distinctive nature of game projects. In our mobile game *World of Tennis: Roaring '20s*, a combination of crash reporting, autobugs and manual bugs, manual testing, playtesting, and automated smoke testing is used. All these elements work together, providing a clear cumulative effect. Most of these subsystems are easy to setup, and can be implemented in a small team on lean budget.

References

[1] [n. d.]. AWS Device Farm. ([n. d.]). https://aws.amazon.com/device-farm

[2] [n. d.]. Crashlytics. ([n. d.]). https://crashlytics.com

[3] [n. d.]. Teamwork.com. ([n. d.]). https://www.teamwork.com

[4] George Amaya, J. P. Davis, D. V. Gunn, C. Harrison, R. Pagulayan, B. Phillips, and D. Wixon. 2008. Games user research (GUR): Our experience with and evolution of four methods. *Game Usability: Advice from the Experts. Morgan Kaufmann, San Francisco, CA* (2008).

[5] Mordechai Ben-Ari. 2001. The bug that destroyed a rocket. *ACM SIGCSE Bulletin* 33, 2 (2001), 58–59.

[6] Bitbar. [n. d.]. Mobile Testing Made Scalable. ([n. d.]). https://bitbar.com/testing

[7] Walter Bright. 2008. Assertions in Production Code. (2008). https://digitalmars.com/articles/b14.html

[8] Jay Fields. 2014. *Working effectively with unit tests.* CreateSpace Independent Publishing Platform, [United States?].

[9] Jez Humble and David Farley. 2011. *Continuous delivery.* Addison-Wesley, Upper Saddle River and NJ.

[10] Ming Huo, June Verner, Liming Zhu, and Muhammad Ali Babar. 2004. Software quality and agile methods. In *The 28th IEEE International Conference on Computers, Software & Applications.* 520–525.

[11] Cem Kaner, James Bach, and Bret Pettichord. 2002. *Lessons learned in software testing: A context-driven approach.* Wiley, New York.

[12] Mohamad Kassab, Joanna DeFranco, and Phil Laplante. 2016. Software testing practices in industry: The state of the practice. *IEEE Software* (2016).

[13] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. What do mobile app users complain about? *IEEE Software* 32, 3 (2015), 70–77.

[14] Taeyeon Ki, Alexander Simeonov, Chang Min Park, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. 2017. Fully Automated UI Testing System for Large-scale Android Apps Using Multiple Devices. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services.* 185.

[15] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. (2015).

[16] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on.* 399–410.

[17] Charles C. Mann. 2002. Why software is so bad. *Technology Review* 105, 6 (2002), 33–38.

[18] Bertrand Meyer. 1992. Applying 'design by contract'. *Computer* 25, 10 (1992), 40–51.

[19] Microsoft Corp. [n. d.]. Guidelines for Smoke Testing. ([n. d.]). https://msdn.microsoft.com/en-us/library/ms182613(v=vs.90).aspx

[20] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2017. Crashscope: A practical tool for automated testing of android applications. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on.* 15–18.

[21] Kevin Moran, Mario Linares-Vásquez, and Denys Poshyvanyk. 2017. Automated GUI testing of Android apps: from research to practice. In *Proceedings of the 39th International Conference on Software Engineering Companion.* 505–506.

[22] Angela Moscaritolo. [n. d.]. Google Play Now Favoring 'High-Quality Apps'. ([n. d.]). https://www.pcmag.com/news/355375/google-play-now-favoring-high-quality-apps

[23] Maxim Mozgovoy, Marina Purgina, and Iskander Umarov. 2016. Believable Self-Learning AI for World of Tennis. In *2016 IEEE Conference on Computational Intelligence and Games.* 1–7.

[24] Maxim Mozgovoy and Evgeny Pyshkin. 2017. Unity application testing automation with Appium and image recognition. In *International Conference on Tools and Methods for Program Analysis.* 139–150.

[25] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. 2014. Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?. In *Proceedings of the 36th International Conference on Software Engineering.* 1–11.

[26] Janni Nielsen, Torkil Clemmensen, and Carsten Yssing. 2002. Getting access to what goes on in people's heads?: reflections on the think-aloud technique. In *Proceedings of the second Nordic conference on Human-computer interaction.* 101–110.

[27] Randy J. Pagulayan, Kevin Keeker, Dennis Wixon, Ramon L. Romero, and Thomas Fuller. 2002. *User-centered design in games.* CRC Press Boca Raton, FL.

[28] Luca Pascarella, Fabio Palomba, Massimiliano Di Penta, and Alberto Bacchelli. 2018. How Is Video Game Development Different from Software Development in Open Source?. In *MSR Conference.*

[29] Steve Sanderson. 2009. Selective Unit Testing – Costs and Benefits. (2009). https://bit.ly/2tVlUcx

[30] Unity Technologies. [n. d.]. Unity Game Development Platform. ([n. d.]). https://unity3d.com

[31] Nishant Verma. [n. d.]. *Mobile test automation with Appium: Comprehensive guide to build mobile test automation solution using Appium.*