QUALITY ASSURANCE IN A MOBILE GAME PROJECT: A CASE STUDY

Maxim Mozgovoy
The University of Aizu
Tsuruga, Ikki-machi, Aizu-Wakamatsu,
Fukushima, 965-8580 Japan
mozgovoy@u-aizu.ac.jp

KEYWORDS

Quality assurance, mobile games, automated testing.

ABSTRACT

Quality assurance is an integral part of the software development process. Game projects possess their own distinctive traits that affect all stages of work, including quality assurance. The goal of this paper is to share the lessons learned during a three year-long mobile game development project. I will discuss the techniques that turned out to be most efficient for us: manual testing, automated and manual runtime bug reporting, Crashlytics crash analysis, and automated smoke testing. I will outline how these types of testing address typical game-specific issues, and why they can be recommended for a wide range of game projects.

INTRODUCTION

Quality assurance is a complex set of methods, used in all stages of software development, ranging from requirements engineering and software design to coding and testing. Explicit quality assurance measures are found in all widely used software development processes, from traditional waterfall model to modern agile approaches [1].

Still, quality issues are common in resulting software products. Khalid et al. [2] analyzed user reviews of 20 most popular iOS apps of June 2012. They found that 26.68% or 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. Indeed, poor design and planning, and somewhat relaxed attitude to writing code is often mentioned as the primary reasons for buggy software [3]. Thus, gradual improvement of software development processes is a necessary, but difficult and time-consuming measure.

I will concentrate on relatively simple, but cost-efficient "last resort" 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.

WORLD OF TENNIS: ROARING '20S GAME

The observations discussed in this paper were made during the development of a mobile tennis game *World of Tennis: Roaring '20s*. The most interesting aspect of the game is the presence of machine learning-based AI system that observes players' behavior to substitute them in player-vs-player matches [4]. This capability allows the players to compete against each other at any time, and mitigates all negative effects of poor internet connection.

From organizational point of view, *World of Tennis* a typical mobile game project, developed by a small team during a time span of three years. The game is written in Unity game engine, and is currently available for iOS, Android, and Universal Windows platforms. The game is free to play (i.e., supported by additional in-app purchases), and requires internet connection for most actions.

GAME DEVELOPMENT-SPECIFIC FACTORS

The nature of a software product we create affects the whole development process, including quality assurance. Game development has its own peculiarities, discussed in literature [5, 6]. The most significant factors that affected our approach to quality assurance were the following.

1. Heavy reliance on unstable 3rd-party libraries and tools.

We have to use specific libraries to integrate with external services (such as ad providers), and to rely on Unity for internal game engine functionality. Some of 3rd-party modules are quite complex, unstable, and may cause app crashes. Often we have to decide whether to use a library that provides a functionality needed for a certain feature, or to cut this feature at all.

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. Diversity of hardware and software platforms.

Unity greatly simplifies the process of cross-platform development, encouraging the developers 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 native binary libraries and in procedures calling platform-specific SDKs (e.g., for in-app purchases).

Diversity of hardware and operating systems also imposes challenges. Some distribution channels such as Apple and Google stores allows the developers to specify the types of compatible devices by providing the required OS version and hardware configuration. It leads us again to treat known flaws pragmatically: if the game does not work properly on certain devices, it might be reasonable to consider them incompatible rather than invest efforts into patches.

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. Similar observations can be made about animation and sound effects.

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 to find nontrivial bugs.

4. Large proportion of high-cost unit testing code.

Literature on agile development speaks in favor of unit testing, but one should note that the associated costs are distributed unevenly. Sanderson [7] identifies two types of code with high cost of unit testing: complex code with many dependencies, and trivial code with many dependences ("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. I 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 dependences" 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 much experimenting, and it is not surprising that the developers tend to view much of their work as "throwaway code", poorly engineered and rarely refactored [5].

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 onscreen 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).

QUALITY ASSURANCE IN WORLD OF TENNIS

This section is dedicated to a more detailed discussion of 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.

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[1]. 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 with inadequate amount of RAM. On mobile platforms, a task scheduler can kill a foreground application if it consumes too much memory, which is practically equivalent to a crash. However, it is hard to decide where exactly one has to draw a line, since 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.

Autobugs and Manual Bugs. Developers widely use *assertions* to check assumptions about certain points in code. Assertions can be seen as a part of "design by contract" approach [8]. There is a general agreement that assertions should be used during development as a method for both in-code documentation and quality assurance, but the practice of keeping assertions in production code is debatable [9]. The arguments often depend on what assertions *actually do*, and the typical presumption is that a failed assertion shows an error message and terminates the application.

In our game, each failed assertion and each raised exception is reported to us. We presume the presence of internet connection on user devices, thus error reporting is easy to automate. Our task and bug tracker Teamwork[2] has a capability to create tasks via email messages, which we use 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 server.

The same technology is used for reporting "manual bugs". 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 started recruiting beta-testers one year before release.

Manual Testing. Our approach to manual testing is 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

[1] https://crashlytics.com

[2] https://www.teamwork.com

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.

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 automated test scenarios [10]. We automate testing of simple routine actions, such as: 1) create a new user and pass the tutorial; 2) play a league match against the next opponent; 3) upgrade 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.

Technically, mobile smoke tests can be set up using an external service, such as Bitbar Testing[3] or AWS Device Farm[4]. 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 [11]. 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.

The scripts interact with our mobile devices via Appium framework[5] 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.

These automated tests can also generate autobugs, so even if there are no obvious faults reported by the test, it still might detect errors via the mechanism of assertions and exceptions.

DISCUSSION

Mobile free-to-play games is a special kind of product. They require long-term experiments with game mechanics, monetization techniques and new features, thus exhibiting the traits of both games and non-game applications.

Research shows that game programmers believe there are substantial differences in their work practice comparing to the work practice of non-game developers [5]. 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 disposable 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 require strict and reliable quality assurance process to make sure that 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.

Therefore, I 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.

CONCLUSION

Numerous objective factors 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, 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] M. Huo, J. Verner, L. Zhu, and M. A. Babar, "Software quality and agile methods," in *The 28th IEEE International Conference on Computers, Software & Applications*, 2004, pp. 520–525.

[2] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?," *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.

[3] C. C. Mann, "Why software is so bad," *Technology Review*, vol. 105, no. 6, pp. 33–38, 2002.

[4] M. Mozgovoy, M. Purgina, and I. Umarov, "Believable Self-Learning AI for World of Tennis," in *2016 IEEE Conference on Computational Intelligence and Games*, 2016, pp. 1–7.

[5] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "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*, 2014, pp. 1–11.

[6] L. Pascarella, F. Palomba, M. Di Penta, and A. Bacchelli, "How Is Video Game Development Different from Software Development in Open Source?," in *MSR Conference*, 2018.

[7] S. Sanderson, *Selective Unit Testing – Costs and Benefits.* Available: https://bit.ly/2tVIUcx.

[8] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[9] W. Bright, *Assertions in Production Code.* Available: https://digitalmars.com/articles/b14.html.

[10] C. Kaner, J. Bach, and B. Pettichord, *Lessons learned in software testing: A context-driven approach*. New York: Wiley, 2002.

[11] M. Mozgovoy and E. Pyshkin, "Unity application testing automation with Appium and image recognition," in *International Conference on Tools and Methods for Program Analysis*, 2017, pp. 139–150.

[3] https://bitbar.com/testing

[4] https://aws.amazon.com/device-farm

[5] http://appium.io