Automated testing: a key factor for success in video game development. Case study and lessons learned.

Christian Buhl, Fazeel Gareeboo
cbuhl@ea.com, fgareeboo@ea.com

Abstract

This paper outlines how the use of automated testing was a key factor in significantly improving the development of a game. We describe our game development process, how we implemented automated testing, and the results of our implementation on key metrics.

Biography

Christian Buhl is currently the Technical Director for the NCAA Football franchise and for the Core Football stability team at Electronic Arts in Orlando, Florida. Christian has a B.S in Computer Engineering from the University of Central Florida and did postgraduate work in Modeling and Simulations at the Naval Postgraduate School. Prior to working at EA Sports, Christian was Lead Engineer for America's Army, the official video game of the United States Army.

Fazeel Gareeboo is a development director at Electronic Arts in Orlando, FL. He currently manages the Development and Release Engineering team, which includes the automated testing group.
Fazeel has a B.Sc. in Computer Engineering from Manchester University and an MBA from Nottingham Business School.

Note on references to colors in this article
Most of the charts in this article have been converted from color to black and white to abide with the printing requirements for the proceedings. The main colors of note in the chart are red and green, and red converts to a darker grey than green.

1 Introduction

EA's Orlando studio (known as Tiburon) develops four major sports video games for Electronic Arts, and this paper outlines how the use of automated testing was a key factor in significantly improving the development of one of those games.

2 Game Development process

The regular development process for our iterative sports titles follows a sequence outlined in the diagram below:

Planning Matrix

	What is the game?	*How will we make it?*	*Prove it!*	*Build it!*	*Prep for launch*	*What's next?*
Planning Areas	Gate 1 **Concept**	Gate 2 **Feature Planning**	Gate 3 **First Production**	Gate 4 **Full Production**	Gate 5 Post Production	Gate 6 **Post Release**

Figure 1 - Game Development Process

The main stages of the game development process that this paper deals with are:

- Full production
- Post production

Full production is defined as complete when QA approves the Alpha build (i.e. the game is feature complete).

2.1 Alpha

Post production comprises of the following phases:

- Alpha
- Beta
- Final

Alpha is typically the most problematic phase, as this is when QA assigns significant resources to testing the game and the game team is in bug fixing mode. Before Alpha, QA staff assigned to the game is usually just a few people, and their role is to help the game team test and tune the game and verify that the game is feature complete. Getting to Beta requires the bug count to be down to 0 non-shippable bugs (crashes, hard-locks, etc).

The constraints of project management are often described as the Project Management Triangle, with Cost, Time, and Scope. In sports video games, the schedule is tied to the season for the real sport, and so there is very little flexibility with Time.

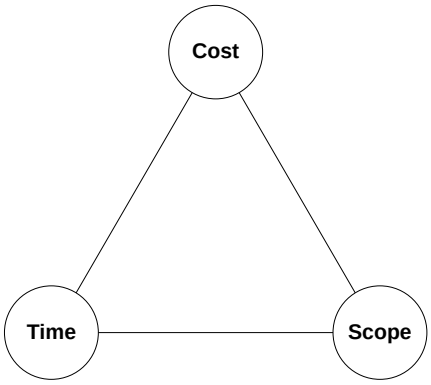


Figure 2 - Project management triangle

Once the game enters Alpha, significant resources have already been invested in feature decisions, and marketing may have already begun, so it is difficult (but not impossible) to reduce Scope. This leaves Cost (developer resources) as the only constraint that can be easily controlled, leaving only a few options in order to hit our dates:

- Increase working hours of existing team (crunch time)
- Add more staff to team

The above adjustments are painful, and hence undesirable. We usually start off with having everyone work extra hours, and if sufficient progress is not being made, we add more staff to the team, and so on.

On previous iterations of the NCAA Football franchise, we typically had poor visibility into the stability of the different parts of the game, which naturally led to poor overall stability. For example, a mode that was not actively being worked on might not be visited for weeks, and when a developer finally needs to work in that mode, it might crash immediately, and the developer has no idea when or how it broke.

When a game enters Alpha with such poor stability, this leads to a high number of defects, and consequently a large increase in hours for the team. This can lead to staff burnout and turnover, which subsequently impacts the quality of the next title.

2.1.1 The search for a good Alpha

In the search for a 'good' Alpha phase, it was evident that we needed to focus on the full production phase, as this was when the defects are originally introduced into the game (Jones and Bonsignour 2012 p188). Specifically, we needed to add process to our Production that would reduce the number of defects that were in the game when we hit Alpha. The Project Manager for the NCAA Football franchise put it this way – "There is no better Alpha. Only better Production".

In NCAA Football 11, we introduced a Quality Assurance Certification (QuAC) process. This is where a developer can ask an embedded QA tester to test their build with local changes. This allows the changes to be validated before checking into the main code repository. Major defects found through QuAC prevent the developer from submitting their changes, and minor defects can be flagged for immediate follow-up. This approach worked well, but takes time (need to push local build to QA tester's machine,

then wait for test result) and is not scalable (we had ~ 5:1 ratio of software developers to embedded QA staff).

QuAC could only get us so far. We had to look for a more scalable solution, and so we focused on improving our use of automated testing.

3 Commit to Stability

3.1 Stop the line

One of the key drivers for focusing automated testing was the NCAA Project Manager, who wanted to implement best of breed test automation. He was influenced by the 'stop the line' literature from Toyota manufacturing (Liker, Hoseus, & Center for Quality People & Organization, 2008). One of the reasons why he was looking into this was regularly hearing the complaint 'Alpha is bad' and 'we are working too many hours' and a hunch that automated testing could be a solution to this problem.

3.2 Support from the Top

Before he pushed for implementing automated testing on the game, the Project Manager first pitched the concept to the Executive Producer, to get buy-in from above. Once this was achieved, he gave clear objectives to the development directors and lead engineers, and made it clear that investing time and resources into automation would be a priority. He also asked for a dashboard to be created that tracked the state of automation.

Once the system was up and running, the Project Manager checked the dashboard every morning for several weeks, and if anything was broken, he would go and ask why and when it was going to be fixed. This made it very clear to the entire team that stability and automation were a high priority from the leadership of the team.

3.3 Creating a Vision

The NCAA Team is divided into several smaller teams called pods. The Core Pod is responsible for non user-facing features and technology, such as memory, technology upgrades, and stability. The Lead Engineer and Development Director for the Core Pod were assigned the task of creating and implementing the policies to make automation work.

The Lead Engineer and Development Director approached stability as if it were a feature of the game. This meant they had to brainstorm it, define the expected outcomes, build a plan to get there, and implement that plan (see section 4 for details of the plan).

3.4 Investing the Resources

With the entire NCAA leadership team committed, the Core Pod was able to invest significant resources into maintaining stability and improving the stability process. Roughly half of the Core Pod's capacity (5 engineers) was spent on stability during the cycle. Additionally, we were able to leverage a large amount of work from the Central Test Engineering (CTE) group. This was a central group setup by the studio a few years ago to lay the groundwork for automated testing of our games.

3.5 Engaging the Team

Finally, one of the most important factors in the success of the stability plan was the way we engaged the team. The NCAA team has a weekly programmer meeting, and we (the Core Pod) used this as a launching pad for the stability process. We communicated the vision for stability to the engineers and

explained how the stability process would benefit them (better Alpha, improved workflow, prevented from being blocked).

Before any part of the process was being implemented, we communicated clearly in advance to the team what it would be, so that everyone knew what to expect. Furthermore, we constantly solicited feedback on what worked/what didn't, so that we could continuously iterate and improve the processes.

Finally, at every programmer meeting, we showed all of our stability dashboards to the team, and discussed any ongoing issues if necessary.

4 Automated testing and its challenges

4.1 Test and Measure It

We began with a pretty simple dashboard of green Checks and red X's. This was a good visual way for everyone on the team to tell what was broken and what was not.

Over time, we added a LOT of detail to the marks (orange triangles for initialization issues, '!' for script loss issues, red diamonds for assert failures), this was mostly useful for the stability team and engineers to help triage issues. In particular, the format of the data was very useful – with CLs on one axis and tests on the other - this way it was very easy to see that a "Script Lost" (screen flow) issue started in the "Dynasty" script in CL 123456. This made it pretty easy to look at the CLs in this range and look for things that were related to Dynasty.

Later on we added even more functionality around here. We have a separate dashboard now that shows asserts, and one that shows crashes.

4.1.1 Automated Testing Infrastructure

The first step to implementing automated testing was to create the infrastructure to support it. We first setup a system that:

- grabs every new game build
- copies it to a game console
- runs a series of test scripts on it
- reports the results to a dashboard

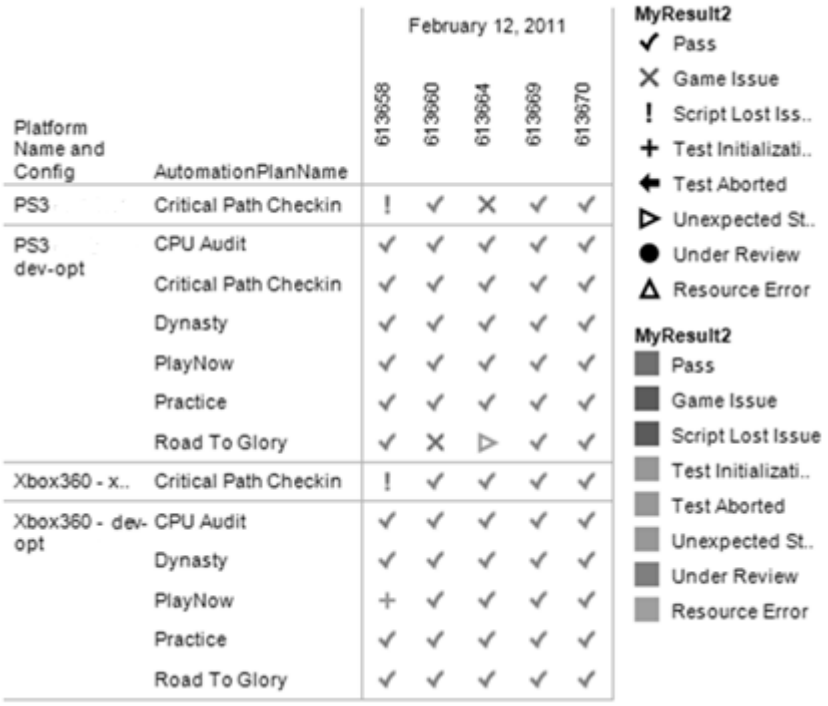


Figure 3 - Game Smoketest Dashboard

The smoketest dashboard shows the results of each script for each game build. The dashboard was accessible to the whole team, and was made to be very visual and color coded, so it was very easy to tell how stable the game was at any time. This took care of regularly testing the main build and reporting on the results.

4.1.2 Bots

We also had a system in place that ran bots every night. The bots are very simple AI scripts that can play games over and over, either solo or in online matches against each other. It tracks how many games completed successfully, how many failed, and detailed information on failures. The bot system:

- Grabs a game build every night
- Copies it to numerous game consoles
- Runs games continuously for a set period of time
- Reports the results to a dashboard.

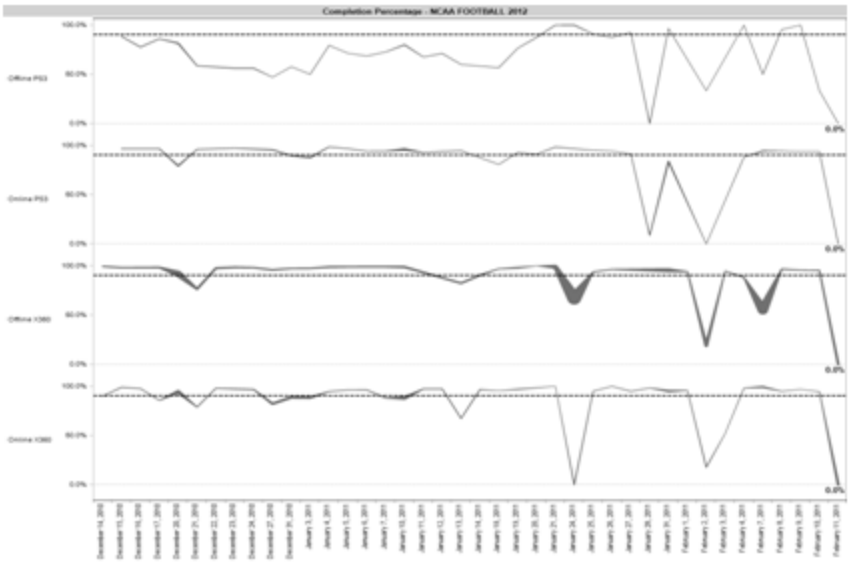


Figure 4 - Bot Dashboard

The bot dashboard shows the percentage of games that passed for each nightly run for each of the four build configurations that we were testing.

4.2 Targets

We set targets for both our smoketest and bot results. We set both a pass percentage target and a recovery time target for each one.

For automation, our target was 100% with a recovery time of 1 day. The expectation was that we would get at least one build every day that passed every single smoketest.

For bots, our target was 90% with a recovery time of 3 days. This longer recovery time was primarily because bots have long turnaround times. We only ran one full bot run per day. If there was a "deep" issue that caused a failure (for example, memory fragmentation after multiple games), it could take hours to triage and debug. It was simply not realistic to expect every bot issue to get resolved within a single run/a single day. Three days gave us three runs to try to get a fix in.

We later updated our dashboards to show this specific information.

The targets were set by the stability lead, who picked the shortest targets that he thought were reasonable.

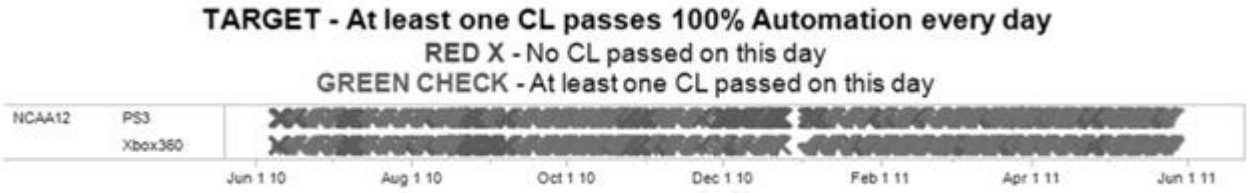
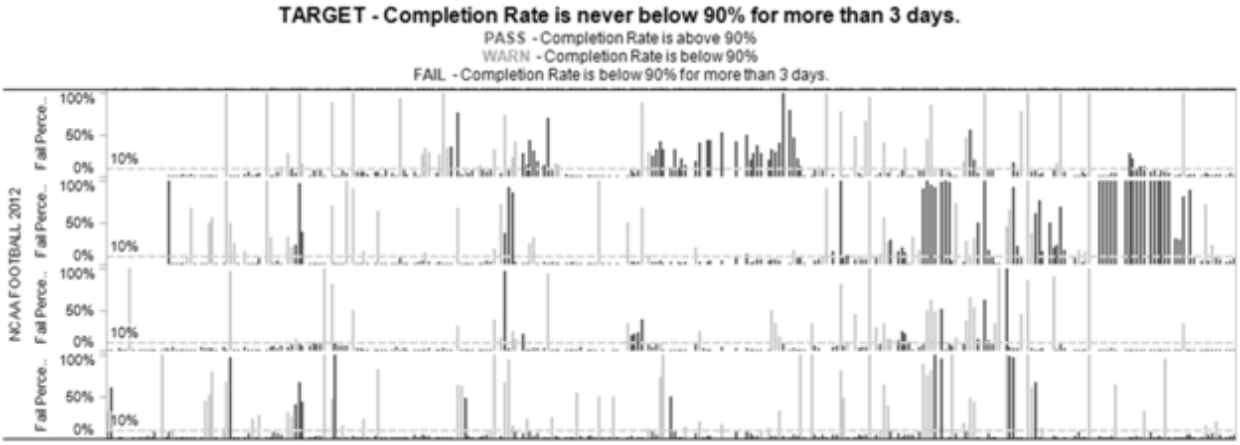


Figure 5 - Game Smoketest Targets



Game Bots Targets

4.3 Prevention

The test infrastructure ensured that we could test and measure the stability of the game. The next step was to make sure that we kept the game stable by preventing issues from being entered into a main build of the game.

4.3.1 At-Desk Automation

We provided every software engineer with the ability to run automated testing at their desk. This was integrated into the tool that software engineers used to build the game to make it easy for engineers to run. The process was simplified so it would take as little time as possible.

We created a "Critical Path" test script that went in and out of all of our game modes (but no deep testing), to ensure that no game mode would be broken. For example, we have a Dynasty mode where the user can control a college football team through multiple weeks and multiple seasons. The Critical Path script entered the "Dynasty" mode of our game, completed the creation stage of a Dynasty, and then immediately exited the Dynasty, without progressing through any weeks. This ensured that Dynasty was never broken at the highest level (creation), but didn't catch deep issues after multiple weeks.
All engineers were required to run *at least* the Critical Path script prior to every checkin. The Critical Path script was run on local consoles, and was kept fairly short (<10 mins) to ensure that it would be convenient for engineers to run.

There was no exact cutoff on what was "too long", this was mostly just a gut call that under 10 minutes was fast enough that it didn't interfere with workflow too much. In many cases, you could just start the script and when you came back to check it after looking at something else it would be done.

Interestingly, after adding this requirement, one of the most experienced engineers on the team, who had a reputation for never breaking things, mentioned that using the Critical Path script was a huge time savings for him. He had previously done a lot of the same testing manually before all of his check-ins.

Together with the 'test automation at desk' feature, we created a report that showed when the main build was broken, which engineers checked in code for that build, and did not run the critical path test at their desk before checking in (CheckInChecker Dashboard). This provided the game team the necessary tools to address 'destructive behavior'.

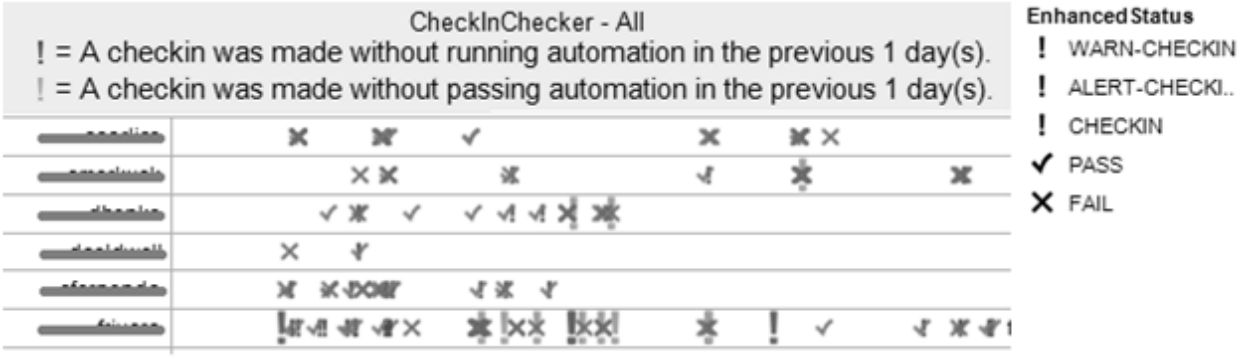


Figure 6 - CheckInChecker dashboard

Getting engineers to run automation before checking in also addressed one of the challenges of automated testing, which is that of making sure the test was in synch with the game. Previously, the test would sometimes fail because the game was changed and the test was failing to navigate the game – as opposed to detecting a failure in the game itself. With the new process, engineers would realize that the test needed to be updated because of their changes to the game, and they would check in the new test with their game code changes.

4.3.2 Expand Coverage

Initially, the Critical Path script was very shallow, it went in and immediately out of a single game, and then in and out of each game mode, without tracking asserts.

As we found things that have caused frequent stability concerns, we have added them to the Critical Path script, keeping in mind that we want to keep it fairly short. For example, our Critical Path did not initially complete a game; it just went in and quit the game immediately. However, we were seeing issues introduced during the End of Game state, so now the Critical Path simulates to the end of a game and exits the game normally.

Most notably, our Critical Path now tracks and fails when it hits an assert. This helps to reduce the introduction of code that is doing bad things, but doesn't actually crash (yet).

We have also added new scripts whenever we have added new game modes, and have expanded other scripts as well.

4.4 Reaction

Having automated testing on each build, and making it easy to run testing at desk, does not mean that the main build will never break. This meant we had to have a policy for reacting to build breakages.

4.4.1 Stop the Line

Our first and most important policy was that if our stability targets weren't met – if we did not get a 100% smoketest pass after a second day, or if we did not get a 90% bot pass after a fourth day, we would stop the line and freeze the depot except for stability fixes – no matter what. Even if freezing work put sprint deliverables at risk, we would do it anyway. The same policies applied to test code. If someone made a change to the Critical Path script itself that was broken, we would back it out. This happened many times, and was just treated the same way as any other issue that broke scripts.

The rationale for this was simple – meeting our stability target was a required deliverable for each sprint, so the sprint wasn't going to pass anyway if we didn't freeze. To reiterate, this obviously requires buy-in and commitment from the team leadership to be willing to put feature deliverables at risk to ensure that stability stays on track.

4.4.2 Back it Out

While the Stop the Line policy was and continues to be our most important policy, it obviously has a very high cost, as we are blocking the work of a large number of developers, particularly if the freeze lasts more than a day (which it usually does).

To mitigate this, we later added a "back-it-out" policy for any smoketest or bot breakage that we could identify. This meant that if a smoketest failed or if there was a significant bot issue, and we could reasonably identify the changelist responsible, we would immediately back out that change. The developer who checked it in would be responsible for resubmitting the change without the offending breakage.

This had the added benefit of impressing on the team the seriousness of game stability, and helped to ensure that all the engineers ran the tests before checking in their changes.

5 Results

The process described above was first tried on NCAA 12 (worked on during 2010-2011).

The results were very positive. Using our own measures of stability, we made demonstrable improvements over the previous year. In NCAA11, our Central Test Engineering team was actually running a small set of smoketests, but without much visibility. The following chart compares NCAA 11 to NCAA 12 across the entire cycle for both titles:

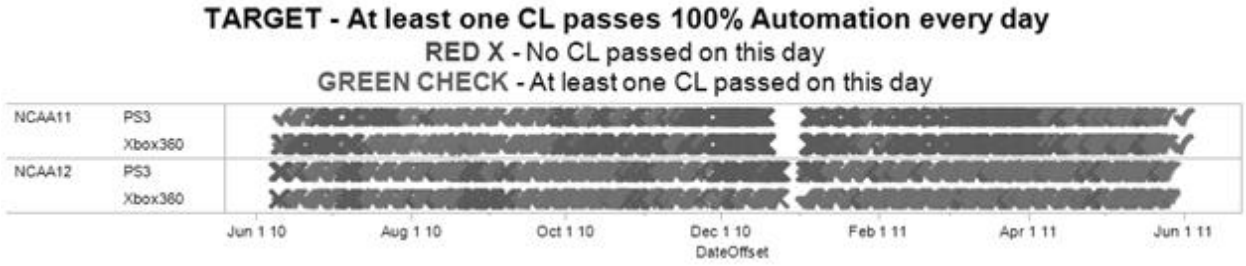


Figure 7 - Smoke Test Stability NCAA 11 & 12

In NCAA11, we went nearly 5 months without getting a passing build. In NCAA12, our worst period was about 3 weeks.

For bots, the picture is similar:

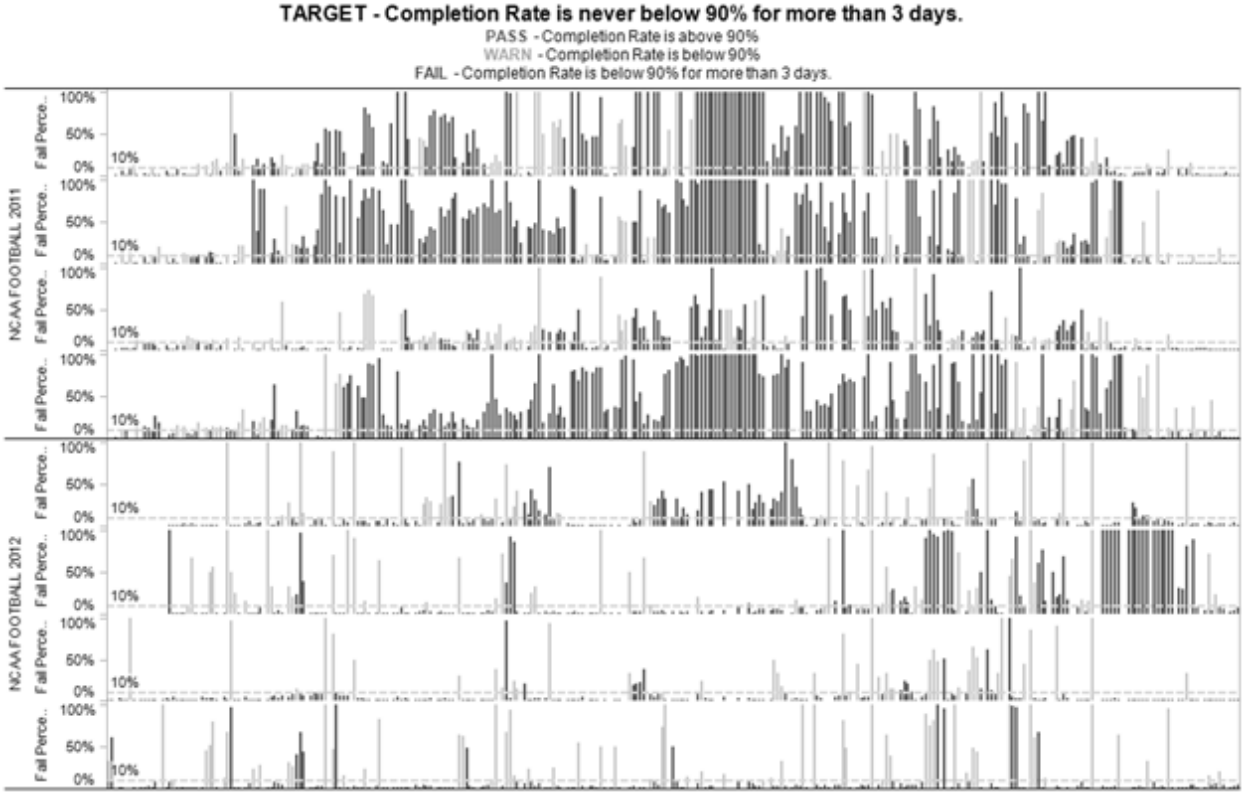


Figure 8 - Bot Test Stability NCAA 11 & 12

In NCAA11, the bots only became stable after we were deep in Alpha. In NCAA12 the bots were stable for most of the cycle. Note that the large red block at the end of NCAA 12 was due to a network outage that was beyond the team's control.

Of course, our real objective was to improve Alpha and the quality of the game. As indicated by this graph comparing Alpha hours between that title and previous iterations, we succeeded here as well. The Finaling team was typically composed of the more senior engineers. As we approached the end of alpha, the Non-Finaling team started to wind down, and all changes in the last few weeks were made by members of the Finaling team.

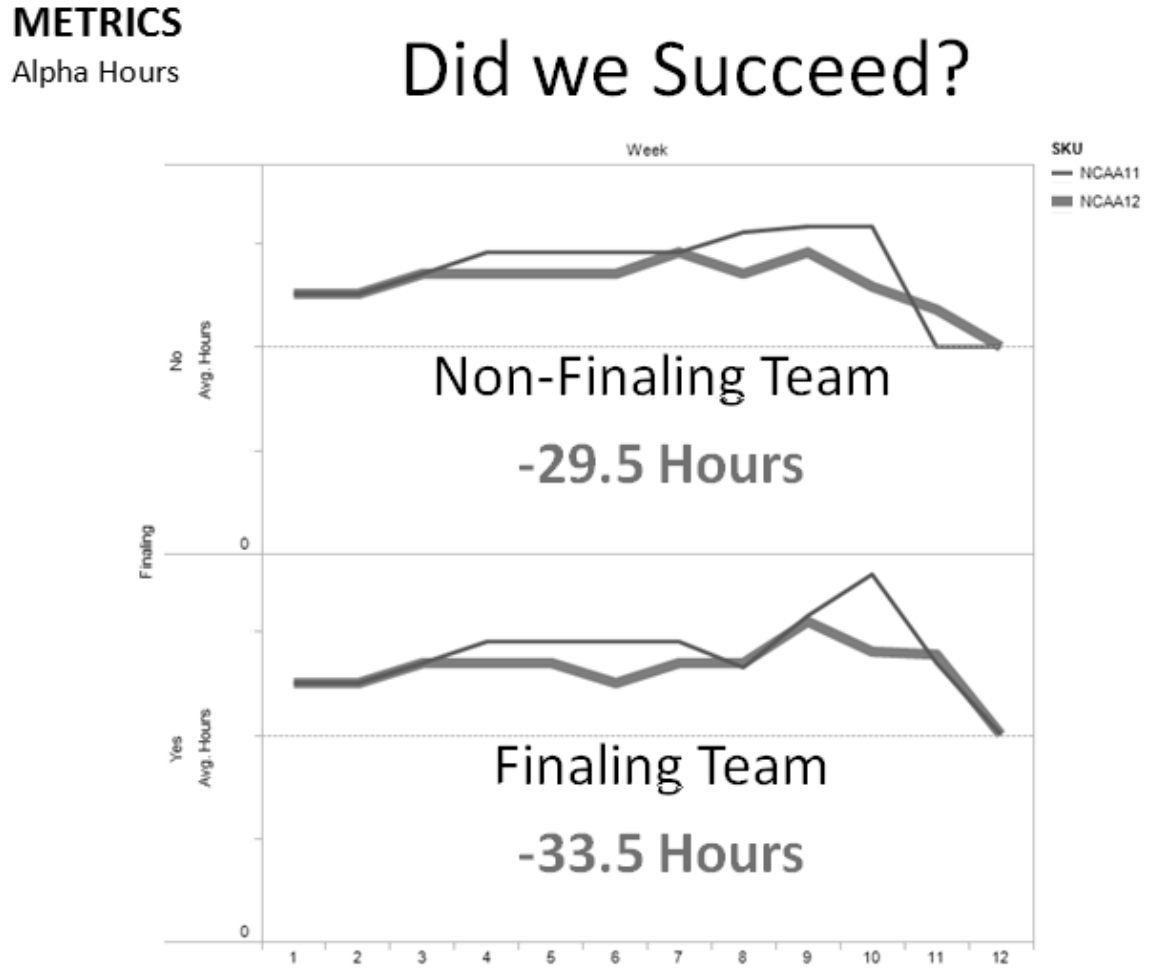


Figure 9 - Alpha Hours NCAA 11 v/s NCAA 12

As you can see from the above graph, both teams ended up working less extra hours during Alpha. This was especially true for the finaling team.

Another way of looking at the results is the graph of Alpha bugs over the entire cycle – again comparing with previous years:

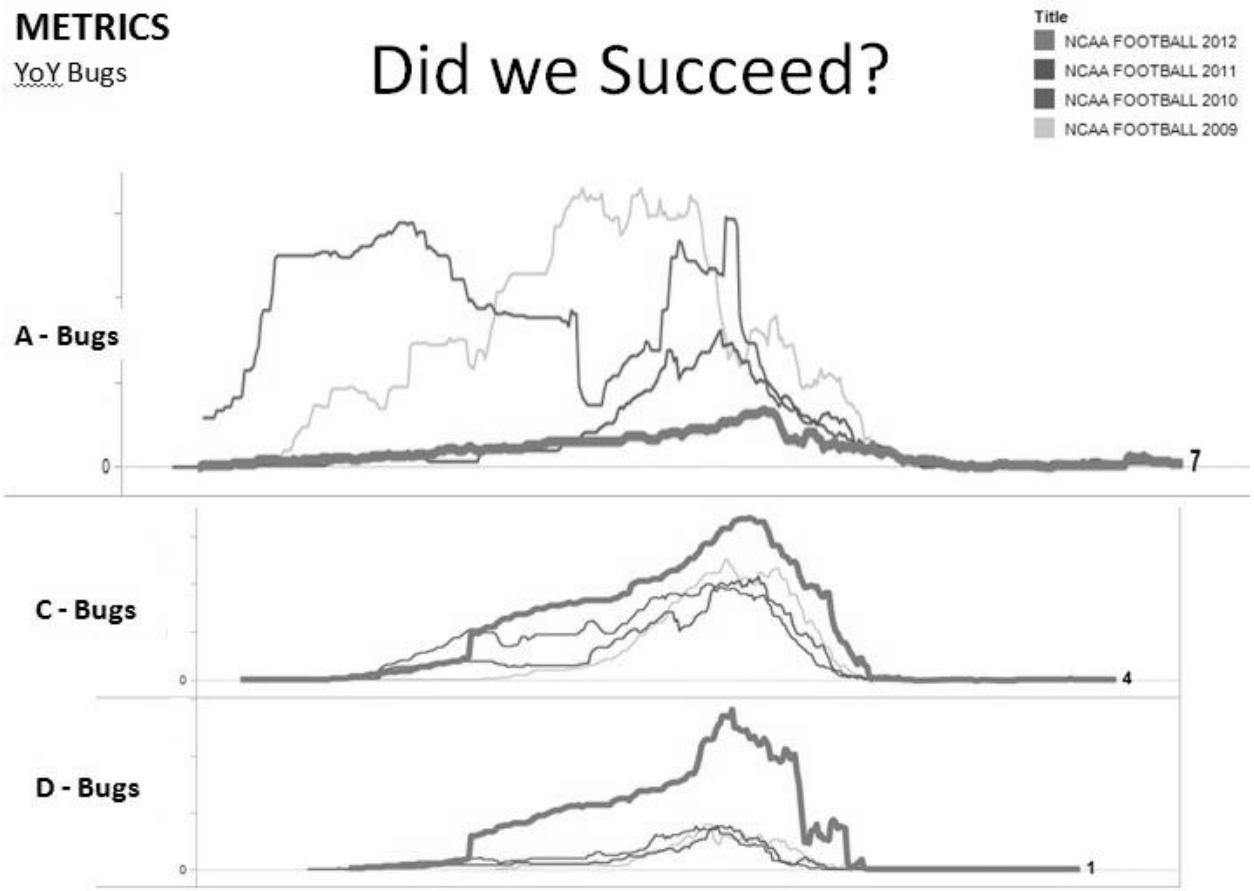


Figure 10 - Year on Year Bugs

Note the drastic reduction in severe (A) bugs from previous years. On the other hand, C and D bugs actually increased (C bugs are defined as medium level – e.g. player names not being sorted in alphabetical order, and D bugs are very minor issues like punctuation). However, A bugs are often blocking bugs that prevent QA from testing areas of the game and finding the lower severity bugs that need to be fixed to make a polished game. By reducing the amount of severe bugs, we ensured that QA had the time and ability to find the lower severity bugs.

We believe that the aspects of the project covered in this paper were all important to achieving the results. If we did not have the infrastructure to support this or the management buy-in or enforced the changes in work habits by taking drastic action (freezing the code-base), this would not have worked.

References

1. Liker, Jeffrey; Hoseus, Michael; Center for Quality People & Organization, 2008. Toyota Culture: The Heart and Soul of the Toyota Way. McGraw-Hill.
2. Rother, Mike, 2009. Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results. McGraw-Hill.
3. Jones, Capers; Bonsignour, Olivier, 2012. *The Economics of Software Quality*. Addision-Wesley.
4. Boehm, Barry; Basili, Victor R., 2001. *Software Defect Reduction Top 10 List*. IEEE Computer, Volume 34, January 2001, p135-137
5. Huizinga, Dorota; Kolawa, Adam, 2007. *Automated Defect Prevention: Best Practices in Software Management.* John Wiley & Sons.