How to write fewer bugs: tips for game developers

freecodecamp.org/news/how-to-write-fewer-bugs-tips-for-game-developers-82e3d742f6f7

2018年11月27日

Stay safe, friends. Learn to code from home. Use our free 2,000 hour curriculum.

26 November 2018 / #Software Development

Richard Taylor

Preface

I have made a lot of games. The end phase of game development is usually hard. The dreaded crunch, the extra mile to fix bugs and add polish can be very arduous for all involved.

If you are lucky the crunch is driven by passion and a desire to achieve a collective vision. If you are unlucky it's driven by over commitment and impossible deadlines. (*In reality luck has nothing to do with it but that is a discussion for another time.*)

The language we developers use when talking about bugs is very telling. Usually something like "QA found a bug in my code!" The implication is that the bug and the code are separate entities, somehow like discovering a caterpillar in some lettuce you were about to eat.

This is, of course, far from the truth — as without the code there is no bug. The lettuce can be removed to leave the caterpillar, the code cannot be removed to leave the bug. Without the code there is no bug. The reality is there are no bugs, just code which satisfies the necessary requirements to a greater or lesser degree.

This language of unaccountably can be very unhelpful when trying to talk about approaches to creating fewer bugs, “What do you mean, bugs just happen, debugging is part of the software cycle.”

In my experience, most game development projects spend a significant amount of resources fixing bugs throughout the project and en-mass at the end of a project. This is essentially wasted time and effort, if these engineers where not fixing bugs they could be adding polish and improving quality.

The answer is clear: Write fewer bugs!

Given that all bugs are just code, then any bugs have been added by the development team. Maybe you should ask your development team to “write fewer bugs”?

If you are laughing at this suggestion I am not surprised. In my experience this is a typical reaction from engineers: no body writes bugs on purpose, right? Yet from a project point of view this is an entirely reasonable request. Imagine the increase in quality or the reduction of overtime if time spent fixing bugs could simply be deleted from the schedule.

The partitioning of code and bugs makes it a very difficult request to take seriously. We are so used to this way of thinking and it seems a nonsense question. So how do we turn “write fewer bugs” into a sensible request which your engineering team can buy into?

Begin new project. Step one, do not repeat previous mistakes.

Our team has worked together on several AAA projects and we were fresh out the end of Driveclub. We all knew we did not want to spend the final months of our next project chasing our tails. Now was the time to do something about it. We wanted to spend time polishing the game not fixing bugs. On starting a new project we sat down with a mind to investigate the idea of writing fewer bugs.

We must develop language and reasoning which will allow the team to take ownership of the problem. It must be clear that the goal is achievable and that responsibility lies with the specific individuals.

Step one is to define ‘why’. We have the top level why: reduce time fixing bugs and spend more time on improving quality. Everyone can buy into less overtime, but why am I asking ‘you’ to fix the problem?

It’s not just less wasted time, rather more time that you are able to spend on the interesting tasks which get you out of bed in the morning. More RnD, more optimisation, more gameplay, more visual fidelity, more excellence. **This is a vision which speaks to why we are all game developers. More time on the cool stuff.**

Even with a clear vision, “write fewer bugs” is so general a statement as to be effectively

useless. It is a statement of intent, but it does not give any indication as to how we might do such a thing. And even more importantly how time invested in “writing fewer bugs” can be turned into a net benefit.

Turning “write fewer bugs” into some kind of technical direction which the engineering team can adopt is going to take significant time and effort. At this point it is all too easy to look at the project deadlines and decide to keep the status quo. After all bugs are inevitable and we have a project to ship.

To discover ‘how’, we follow the basic scientific method: **Observation, Hypothesis, Tests, Repeat.**

Observations

Categorisation — Forensic analysis

Open the bug data base from your last completed project and have a look. There are many different types of bugs, so to just say fewer bugs is essentially meaningless.

To be able to discuss the problem we first needed to better define the specifics. In this case, we were particularly interested in bugs which soak up significant amounts of engineering time. We analysed the bug database and identified 2 significant cohorts we wanted to tackle:

- Bugs that took a very long time to fix, and
- bugs which continually regressed.

Essentially 2 KPI values for bugs: ‘days to fixed accepted’ and ‘number of bug state transitions’.

Bugs with a long time to fix are usually bugs which are difficult to reproduce or diagnose. Bugs with many state transitions are often bouncing around ‘cannot repo’ or ‘fix / reopen’ loops.

Using these KPIs we identified a smaller set of bugs which account for a dis-proportionally large amount of engineering time. We had a few false positives in these cohorts which were discarded in the next step. Finally we had our way into the problem.

Analysis

Having identified specific cohorts of bugs, the next step was to try and identify commonality and root cause. This required a combination of programming experience and interpretation of bug description. Most of all it required time and perseverance.

We had integration between our bug database and SCM. This allowed some direct correlation to source code, although the noise ratio was high and experienced interpretation still required.

Root Cause

Eventually, after enough time, some patterns did emerge and we could see that a subset of bugs did indeed account for a significant amount of engineering time. Given these bugs, we were determined to get to the root cause, to the source code.

Working with the engineers we found the 'fix changes'. We then compiled more lists of systems, files and lines of code which were related to the significant issues. Ultimately, we had a list of specific code changes to discuss with the engineering team.

Told you so

Then we were able to sit down with the wider engineering team to discuss our findings (and they were almost certainly aware of all the issues in the code!!!). So if the team already knows about the issues in the code, what have we accomplished? The answer is something very important.

We have created a mapping between 'lost development time' and 'specific areas of the code'. This gives us a way to objectively rank the value of any proposed refactoring and maintenance.

It also serves to highlight areas of code which have become 'acceptably bad'. This was our biggest surprise. I found myself saying 'we clearly need to refactor this system' when the engineering team had already ruled it out as '*too big a job*' or '*just not feasible*'.

Indeed, many of the root cause problems were systemic. In particular, many of the 'unseen' issues involved what were generally accepted programming patterns. **This means that fixing these issues would require defying current programming trends, fashions and mantras.**

(Ultimately these root causes where so systemic in the code architecture that it led to us starting again entirely from scratch. But that is a topic for another blog post.)

Finally we had enough observations to make a hypotheses and the whole team has been involved in the journey.

Hypothesis

> **Hypothesis 1** — Specific programming patterns will be statistically more likely to cause bugs in a large software project.

> **Hypothesis 2** — Time spent fixing bugs throughout the project will be reduced if we avoid the use of programming patterns identified by (1).

For the purpose of this article, let us define a large project as 25+ programmers and 12+ months development. A project large enough that the following holds true:

a) Any code will live long enough that cost of maintenance will outweigh cost of development.

b) The complexity arising in the glue between systems is greater than the complexity of any single system.

Why is this significant? In small software projects you can get away with anything, software engineering basically does not matter. The code is all yours.

In a large project the code is NOT yours. You will work with code you do not understand and will base engineering decisions on imperfect knowledge and assumptions. Now when we talk to the rest of the team the message is very different.

“Having analysed our previous projects we have made the following hypothesis….”

“The data shows that these specific programming patterns were a common factor involved in the issues. We believe avoiding these patterns will reduce the time spent fixing bugs and improve quality.”

Next we need to turn the hypotheses into something usable.

Test

An important part of the technical direction and systems architecture for the next project would be based on avoiding identified high risk code patterns.

- Coupling memory allocation lifetime with object construction and lifetime.
- Overloading operators and de-normalised naming conventions.
- ‘auto’, polymorphic functions and the removal of type safety. (https://medium.freecodecamp.org/why-the-compiler-is-your-best-friend-f165329cb20a)
- Pushing complexity upwards by dependency injection, callbacks, lamdbas,
- Using mutex, semaphore and other threading primitives in high-level code.

Each of these points is worthy of technical discussion about how it changed our approach to systems and API design. As this technical discussion would be for a slightly different audience, I will address them in separate technical follow-up posts.

What happened next?

As I mentioned above we were extremely lucky and had the opportunity to start a new game with a new approach. We were able to build a new game engine from scratch in 24 months and deliver the game on time with a small programming team. Even though the code had never shipped before we achieved a high level of polish with relatively few bugs and very few high cost bugs. This was independently confirmed by the QA department who documented the easy time the game had through the QA process.

Asking the team to avoid the patterns above was not enough. Coding guidelines are easy

to forget and the team could have quickly slipped back into old habits. A key decision was to design the code, systems and interfaces such that the above patterns could not be used. This quickly developed into a mantra “make it hard to do the wrong thing” which guided the team through the entire project.

Importantly the team was happier. We had defined an approach which resulted in more code working the first time. The team knew that the obvious and easy way to use a system was probably correct and that the API would prevent bad patterns. There were fewer bugs and more time spent on features, polish and iteration.

Further reading.

You can check out these annotated slides containing specific technical direction derived from the Write Fewer Bugs approach.

https://www.slideshare.net/RichardTaylor172/c-restrictions-for-game-programming

Richard Taylor

Read more posts by this author.