Defect Management by Policy: A Fast Easy Approach to Prioritizing Bug Fixes

by Mike Cohn •

20 Comments

Image not readable or empty
Defect Management by Policy: A Fast Easy Approach to Prioritizing Bug Fixes

Long before the world discovered agile, prioritizing bug fixes was a challenge in software development. But agile's short iterations make it even harder for many teams to decide which bugs to fix and which to put off. The good news is, an agile team typically has far fewer bug fixes to sift through than teams using more traditional software development frameworks. But most agile teams still find a few bugs along the way, especially if some of the development was done prior to the team adopting an agile approach. And they need an efficient way to prioritize those defects.

Prioritizing Bug Fixes: What Not To Do

Early in my career as a programmer, my boss had our entire team spend a week going through every known defect. We discussed possible causes, the severity of each bug, how often it was occuring, whether the bug had been reproduced, where in the code was the bug likely to be, and which of us should fix the problem.

We even estimated how long each fix would likely take. Not only were those estimates largely worthless but in plenty of cases, it took longer to estimate a fix than to just do it.

Having seen early the wasted effort that this caused, when I began leading teams, I started experimenting with a few other, lighter approaches.

I want to share my favorite with you today.

Prioritizing Defects by Policy: A Better Approach

Rather than taking time to reflect on each new bug individually, establish **defect policies** that

determine how quickly a bug should be fixed.

One defect policy might be that any bug affecting all users in a dramatic way gets fixed immediately, meaning it interrupts work in the current sprint. Another defect policy may be that a bug that occurs only in extremely rare circumstances and doesn't prevent a user from completing critical tasks gets logged and is fixed whenever time permits but without any urgency.

Creating and using policies this way is known as *prioritization* by policy.

As a more specific but obvious example, a team and product owner might agree that any bug that is preventing orders from being submitted on their eCommerce website needs to be fixed ASAP.

Other policies may define bugs that need to fixed by the end of the day, the end of the week, or not at all.

Define the Defect Policies

A useful way to formulate bug-fixing policies is by considering:

- Defect Likelihood: How often will the problem occur?
- Defect Severity: How bad is it if the problem does occur?

Notice that I refer to these as *likelihood* (or *frequency*) and *severity*.

Consider a hypothetical bug on Amazon.com that orders over $1 million are not being submitted because a developer made an assumption that orders would never exceed $999,999.99.

This is bad when it occurs (high severity), but I have to imagine Amazon doesn't get a lot of orders that exceed $1 million (low likelihood).

Create a Defect Policy Matrix to Prioritize Bugs

The two dimensions--severity and priority--can be combined to establish the priority policy for the defect. To do this, create a simple matrix cross referencing those two factors as I've done here:

Likelihood:

Severity:	< 1% OF TRANSACTIONS	1% OF TRANSACTIONS	< 10% OF TRANSACTIONS	> 10% OF TRANSACTIONS
Easy, obvious workaround available	Very Low	Low	High	High
Non-obvious workaround available or workaround available only for some users	Low	Medium	High	Very High
Important functionality unavailable	Medium	High	Very High	Very High

There are many ways you can categorize likelihood and severity. Sticking with an eCommerce site example, I used the percentage of transactions affected for likelihood. Anything estimated to affect 10% or more of transactions is a pretty big deal, so I've set that entire column to High or Very High.

For Severity, I used whether a workaround was present or not and obvious or hard to find. On an eCommerce site perhaps there are two "Buy Now" buttons and only one is working.

The cells in the matrix indicate what policy should be in effect for defects of the indicated likelihood and severity. In this example, I used five priorities from Very Low to Very High. In some cases you can get by with as few as three. I'd be cautious of needing more than five, although I have seen it.

Here's how I will commonly use a five-item set of policies:

Very High: Added immediately to the current iteration even at the risk of delaying that work. May very likely need the effort of more than one team member, possibly including the whole team.

High: Added immediately to the current iteration even at the risk of delaying that work. Team decides who can best address the issue.

Medium: Added to the current iteration at the discretion of the product owner.

Low: Documented. Discussed in the next iteration planning meeting at the discretion of the product owner.

Very Low: Documented in list of known issues. Revisited only if severity or likelihood increases or at the discretion of the product owner.

Advantages of Prioritizing Bugs by Policy

The key advantage to this approach is that it greatly reduces time spent debating what should be done with each defect. Prioritizing bugs by policy does require some initial effort to create the right policies. But once those are created, prioritizing each new defect report becomes nearly trivial.

The goal with an approach like this is that prioritizing defects becomes largely objective rather than subjective. Yes, someone will have to decide how frequently a problem will likely occur, but beyond that prioritizing defects becomes no more time consuming than consulting the team's policy matrix.

What Do You Think?

What do you think about prioritizing defects by policy? What steps has your team taken to make prioritizing defects easier? Please share your thoughts in the comments below.

Posted: April 10, 2018

Tagged: defect management, prioritizing

About the Author

Mike Cohn specializes in helping companies adopt and improve their use of agile processes and techniques to build extremely high-performance teams. He is the author of User Stories Applied for Agile Software Development, Agile Estimating and Planning, and Succeeding with Agile as well as the Better User Stories video course. Mike is a founding member of the Agile Alliance and Scrum Alliance and can be reached at hello@mountaingoatsoftware.com. If you want to succeed with agile, you can also have Mike email you a short tip each week.