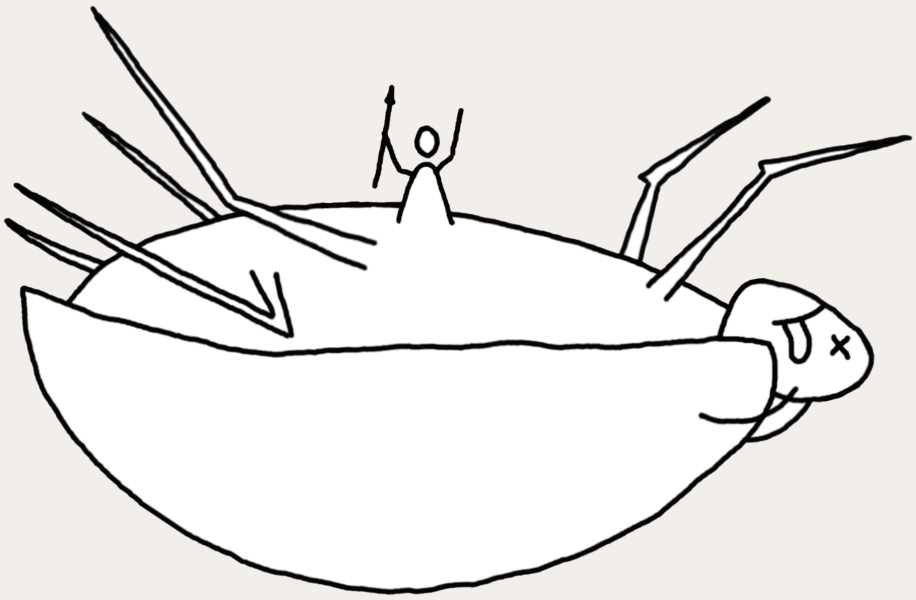
PROJECT INSIGHTS



ZERO BUG POLICY

A PERSONAL FIELD REPORT

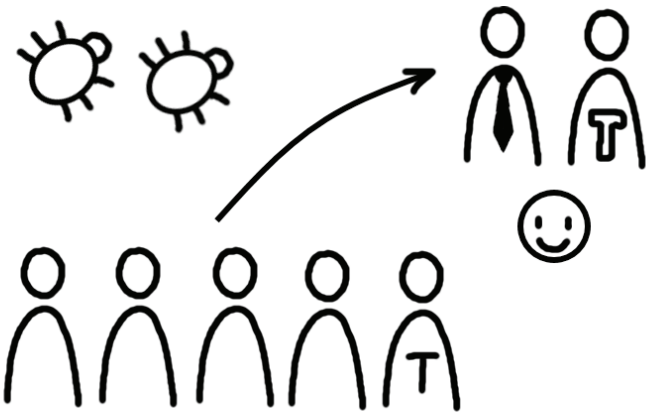
An alien planet. An outpost of humanity. It's quiet. Then someone shouts: 'Bugs! Bugs!' Soldiers run to the fortification walls. The camera follows them. Behind the walls, the audience sees a wasteland with hills in the distance. Then the first wave of an alien ambush pours over the hills. It is a swarm of horrifying insectoids. Bugs. More and more bugs scale the hills. Seconds later, thousands charge towards the outpost. The soldiers fire desperately into the masses but there are just too many attackers. The bugs reach the fortification walls, pile up and finally overrun the soldiers.

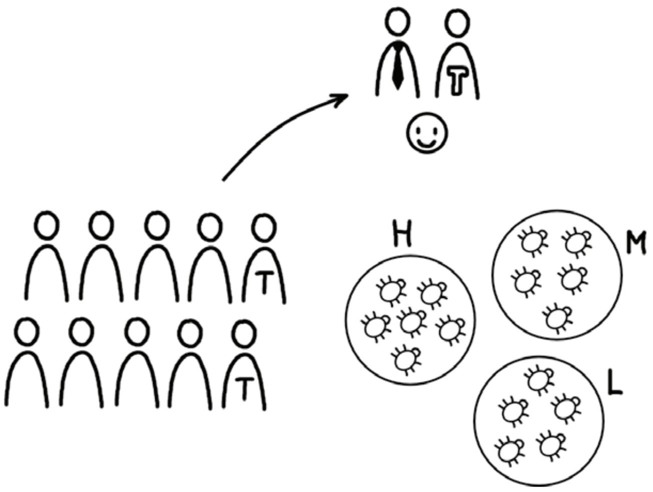
If this sounds familiar, you might remember this scene from an old science fiction movie [1]. Or you might be working in software development.

What follows is the story of a software project that found itself in a similar situation and survived.

A STORM IS BREWING

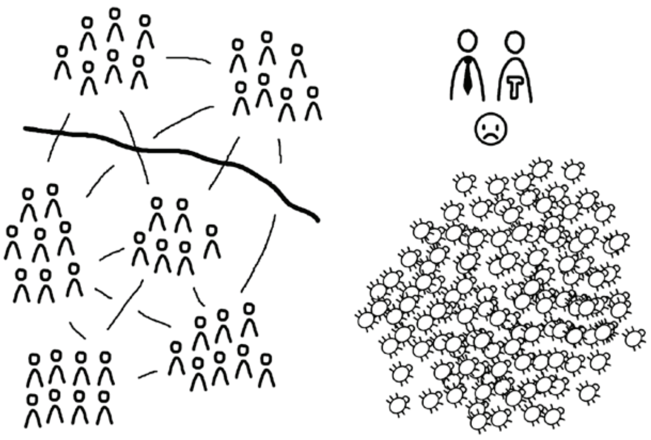
In the beginning, everything was fine. We were a small Scrum team with our very own testing specialist, developing software for a new hardware product. Unfortunately, we were not able to work directly together with our future customers. Instead, we delivered new versions of the software in small iterations to a company-internal testing department. At that time, we had the bugs under control, and the company was happy with the delivered quality.





Then we were joined by a second team. The two teams still delivered good quality, but at the same time, we started to have more bugs. To simplify the planning, we grouped the bugs into the priorities high, medium and low. We tackled high-priority bugs immediately. Medium ones were fixed when we had some spare time (almost never), and the bugs with low priority weren't even on our radar.

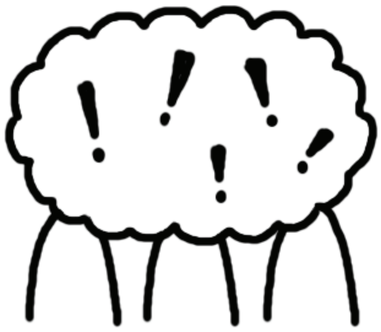
Some time later, unexpectedly, a large set of new requirements was added to the project. To prevent delays, four additional teams joined the development force, two of them from other countries. The software architecture, however, was still geared towards development in a single team. We were therefore constantly getting in each other's way and the number of bugs went up into the hundreds.



UNDER SIEGE

With this new situation came a lot of problems. The developers lost all hope of bringing down the overwhelming number of bugs any time soon. Why care about a new bug? It was just one of many.

Instead of fixing the bugs, we spent a lot of effort on workarounds. At that time, preparation for a software demonstration usually went something like this:



First, install software on target platform … then prepare data for the demonstration … check functionality just to be sure … damn, it's not working … why … um … if I try this … no … and that … no … let's ask the others … ah, this is the well-known bug X … as a workaround, I have to manually update a file before installation … ok, let's start over… great, now it's working … so let's just do a fresh installation, so I have a clean system … prepare the data for the demonstration … ready.'

A little later during the demonstration:
'Let's demonstrate the new functionality … damn, it's not working again … oh, I forgot the workaround, when I did the installation the second time … let's start all over again … sigh …'
On top of this, we invested a lot of time in bug management. Once a week, the bugs were prioritised and assigned to the teams. It often went like this:

1st week: The bug management team reviews bug X. The team does not understand the bug description and sends a request for clarification to the reporter.

2nd week: The bug management team reviews the same bug X, now having feedback from its reporter. The bug is assigned to team A.

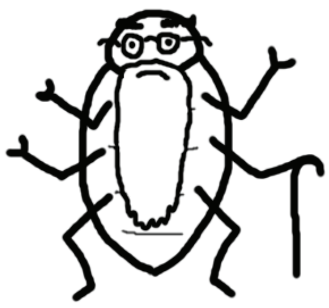
3rd week: Team A examines bug X and sees that the assignment was wrong. The bug is returned to the bug management team.

4th week: The bug management team reviews bug X. It looks awfully familiar. With a strange sense of déjà-vu, it assigns the bug to team A.

5th week: Team A examines bug X and realizes that it was assigned to the wrong team for the second time. To prevent this from happening again, team A talks directly with someone from the bug management team.

6th week: The bug management team reviews bug X and assigns it to team B.

7th week: …and so on…

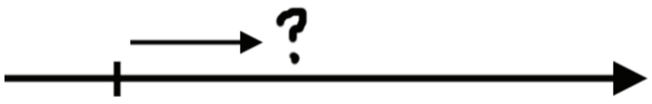


Owing to such bug management and prioritisation, some bugs could become quite old before they even reached a developer. If you then wanted specifics about the bug, the reporter often couldn't even remember writing it.

If the bug was not reproducible in the current version, you had a difficult decision to make. Should you assume that the bug was fixed coincidentally by recent changes to the code? Or should you go back to an older version, so that you could try to reproduce the bug?

The investigation of such ghost bugs often involved restoring an old image on the target platform or taking old hardware from a storage room, which could become very expensive.

For project planning, the situation was also very unfortunate. It was very difficult to plan bug fixing. Supposedly simple bugs became real monsters, while other, seemingly more complex bugs were surprisingly quick and cheap to fix. Under these circumstances, it was almost impossible to predict when the software was going to be finished.



ZERO BUG POLICY

No one was happy. We all wanted to change the situation. But how? At that point, someone recalled a statement they had heard during a Scrum Master certification course: 'No bug survives the night.' This seemed quite extreme, but what did we have to lose?

To that end, we tried to convince the project lead to adopt a Zero Bug Policy. We proposed pausing work on new features until we'd carried out all the bug fixes necessary for the product launch. And we were lucky; in the end, we had the full backing of the project lead.

At that time, we just got cracking. If only we'd taken the time to check if there were others out there who'd already had experience with such an approach, we could have prevented a major setback along the way. But first things first – here's the definition that could have helped us back then:

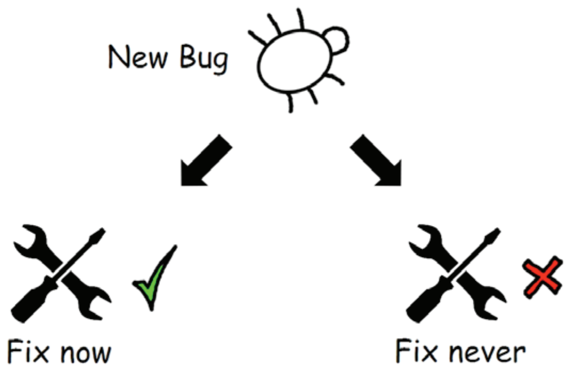
Zero Bug Policy
No known bugs

Beware! It does not say 'No bugs'.
It says 'No *known* bugs'.

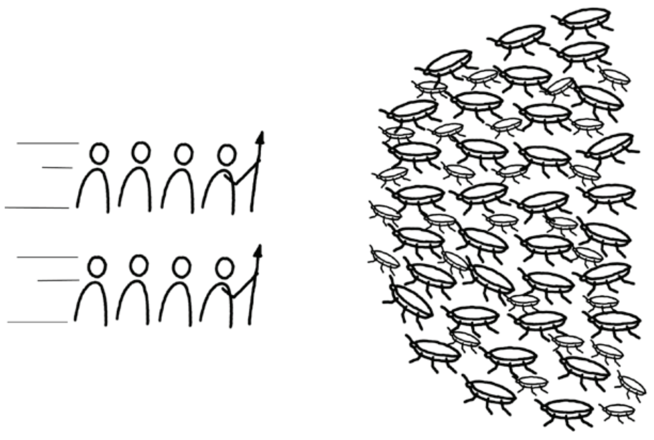
The Zero Bug Policy is sometimes also known as *Zero Bug Tolerance*. This often leads to the mistaken assumption that you're setting out to achieve a completely bug-free product. But if your product has a certain complexity, this is no longer economical. The Zero Bug Policy only affects known bugs; it does not specify how much effort we ought to expend in finding additional bugs when – at a certain point – there are no known bugs in your product.

Does this then mean that you must fix every known bug? Yes and no. The Zero Bug Policy has a very simple approach. When you find a bug, you only have two options:

- You fix the bug immediately.
- You never fix the bug.



Just to be clear: if you decide not to fix a bug, there is also no management or tracking of it. It is, by definition, no longer a bug, but rather a product behaviour that is deemed acceptable. This sounds extreme, but it is necessary for the Zero Bug Policy to develop its full potential.

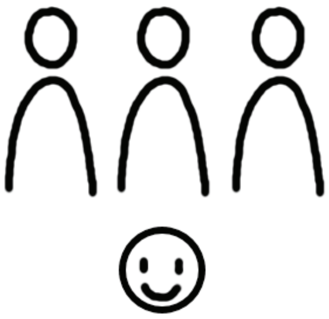


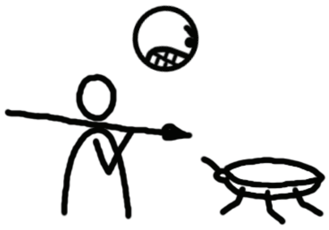
When we first came up with the idea of the Zero Bug Policy, we were unaware of this definition. Our motto was just: 'No bug survives the sprint.' To that end, we first had to bring the existing bug count down to zero. Again, we were lucky, and the project lead agreed to defer work on new functionality in favor of this goal. We could dedicate all our efforts towards the bugs. To reduce the total effort, the bug management team rejected all existing bugs that were not relevant for the product launch. It took more than one sprint, but we reached our goal in the end: no known bugs in the system.

THE EFFECTS OF ZERO BUGS

The next goal was to keep the bug count at zero. If new bugs were found during a sprint, we fixed them at the earliest opportunity. The teams had different ways of tackling this. One team created a rule that developers had to fix at least one bug before beginning a new story. Another team chose one person per sprint who focused primarily on bugs. Yet another team coordinated the bug fixes during the Daily Scrums.

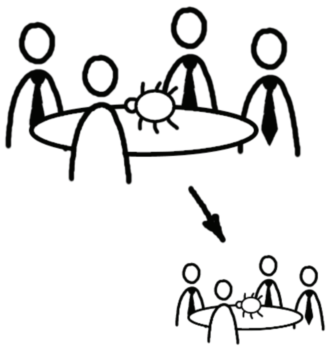
Regardless of how the teams put the Zero Bug Policy into practice, it didn't take long before remarkable changes began to happen. First, the sense of despair suddenly vanished. It was as if a huge weight had been lifted off our shoulders..

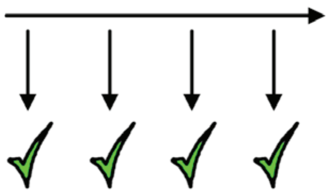




Before the Zero Bug Policy was implemented, a new bug used to be just one of many. Bug-fixing was a necessary chore; something standing between you and the exciting work. With the Zero Bug Policy, on the other hand, a new bug resulted in dented pride on behalf of the team. Some developers even got carried away with bloodlust. The Scrum master had to ensure that they did not drop everything else each time a new bug was found.

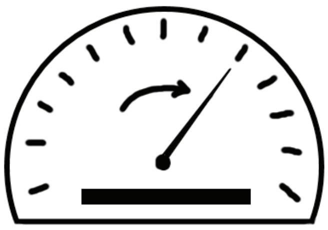
The bug management did not disappear, but it did get easier. Take the duplicated bugs, for example. With hundreds of bugs, it is rather time consuming to find duplicates. With the Zero Bug Policy, it's a piece of cake. Usually, you'll have zero bugs. So, there's no real need to look for duplicates in the first place. Even with, say, five bugs, it's a trivial problem.





The Zero Bug Policy helped to improve the results of our sprints as well. An important goal of a sprint is to produce a potentially shippable product. With hundreds of known bugs, the versions we produced in earlier sprints were far from being deliverable. Although the Zero Bug Policy alone did not suddenly make our product shippable, it enabled us to take a giant leap towards our goal.

Beyond that, we were also able to see improvements in the collaboration with the testing department. Before the Zero Bug Policy, the delivery of a sprint went a bit like this: 'This is the next version of the product. Find the twenty new bugs we added to the hundred known ones during the last sprint.' With the Zero Bug Policy, our attitude changed to: 'This is the next version of the product. There are no known bugs. Prove us wrong.'

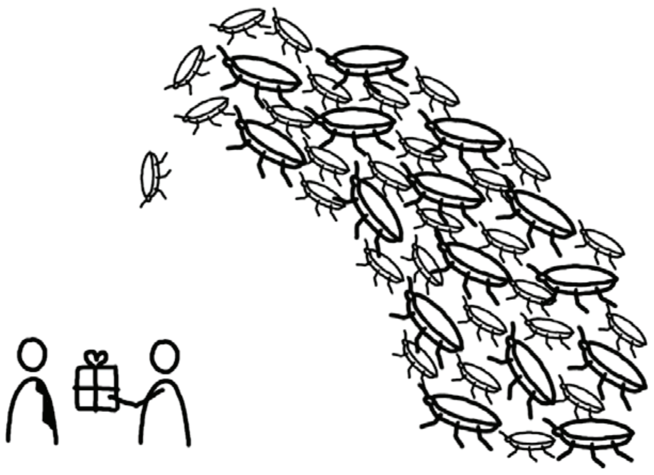


The Zero Bug Policy had yet another effect. And this effect alone justified the initial effort and all the risks which came with such an extreme approach. The velocity of the teams increased by at least 30% with the introduction of the Zero Bug Policy. I can't say for sure what this jump is down to. Was it the motivational boost of all those involved? Was it because we were not wasting time on workarounds? Was it the reduced bug management? Did the product just have a better quality in general? Was it a combination of all these factors? Or was it something completely different?

At the end of the day, it doesn't really matter. What's important is that the Zero Bug Policy helped us to drastically reduce the amount of work that did not add value to the product.

IS IT DEAD?

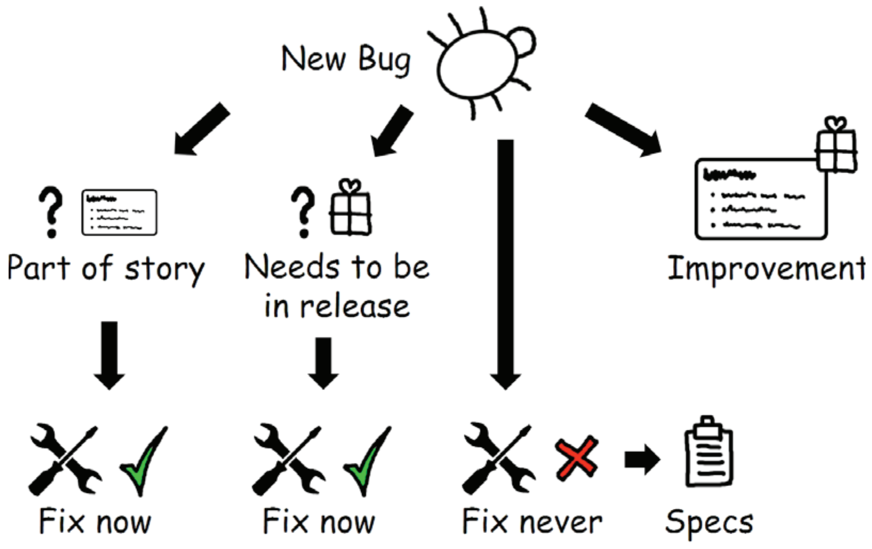
This is our happy ending, right? Unfortunately, no – not really. Like so many Hollywood stories, the monster everyone assumed was dead reared its ugly head one more time.



Version 1.0 was finished. This was the first version we delivered to an actual customer. As the development teams began to work on the next product version, something unexpected happened. The bug count jumped suddenly from zero to a high two-digit number. What had happened?

We had overlooked an essential element of the Zero Bug Policy. Instead of deciding whether a bug should be fixed or not, the bug management team had simply shifted all bugs that were not relevant for version 1.0 into a future version.

Is this wrong? Does the Zero Bug Policy really require that you make a decision every time about whether or not to fix bugs immediately or not at all? We were looking for an answer to this question when we found a more detailed definition of the Zero Bug Policy.



First, ask yourself: does the bug belong to a story of the current sprint? If so, the story is not yet done – the bug needs fixing immediately. So far, so obvious. Unfortunately, we had teams that closed stories with open acceptance criteria by creating corresponding bugs.

The second question you already know from the simpler Zero Bug Policy: does the bug have to be fixed before the product is delivered to customers? If so, fix it immediately.

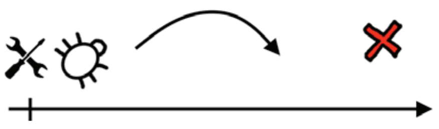
If the bug fix is not needed for the next release, but would substantially increase the value of the product, it is no longer considered a bug. Instead, it is treated as an improvement and included in the planning for future releases as a story. The initial bug is removed from bug tracking. For the current version it is considered an expected behaviour.

Finally, you can still decide to never fix a bug. This is usually the case if the bug occurs infrequently, represents a very low risk, would be very expensive to remove and has little impact on the added value of the product.

Regardless of whether you never intend to fix the bug, or whether it is moved as an improvement to the next product version: the bug ceases to be a bug and becomes an expected behaviour. Specifications should reflect this. If you generate your specifications from automated acceptance tests, this is easy. Just add another test for the behaviour.

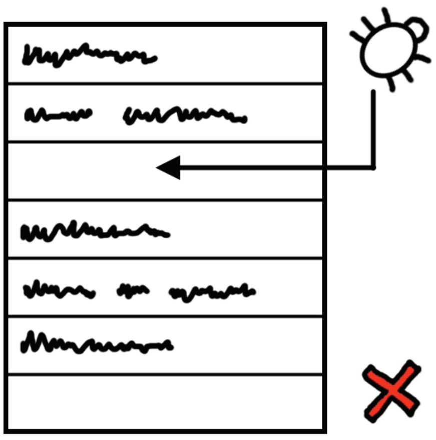
WHY NOT...?

'Why do I have to fix a new bug immediately? We have much more important things to do right now! Why can't I just finish my stories and fix the bug in the next sprint?' When we started using the Zero Bug Policy, we heard such questions a lot.

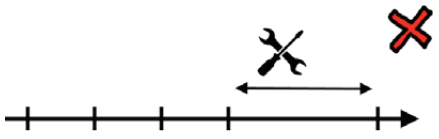


But it turned out there were no real advantages to postponing the bug fixing to later sprints. Every sprint brought with it more important work and, on top of that, new bugs. 'I'll do it later, when I have more time,' didn't really work for us.

Before the Zero Bug Policy, we tried to put bugs into the backlog. Unfortunately, the effort involved in fixing a bug was very difficult to predict and our backlog began to fluctuate severely. To make things worse, our product owners were not particularly interested in bugs. If something needed to be dropped from a sprint backlog, bugs walked the plank first.



For a while, we had the plan to fix most of the bugs during a stabilisation phase at the end of development. Unfortunately, by that time, a lot of bugs would already have been very old, which would make them more expensive to fix. It also poses a substantial risk for the release plan. Let's assume that a bug fix introduces a new bug with a probability of 30%. With 300 known bugs which we manage to fix in the stabilisation phase, we get about 100 new bugs. It is quite likely that these new bugs contain show-stoppers, which would delay the release.



With the Zero Bug Policy, there are no known bugs as you enter the stabilisation phase. Therefore, you don't run the risk of getting any additional show-stoppers as a result of late bug-fixing.

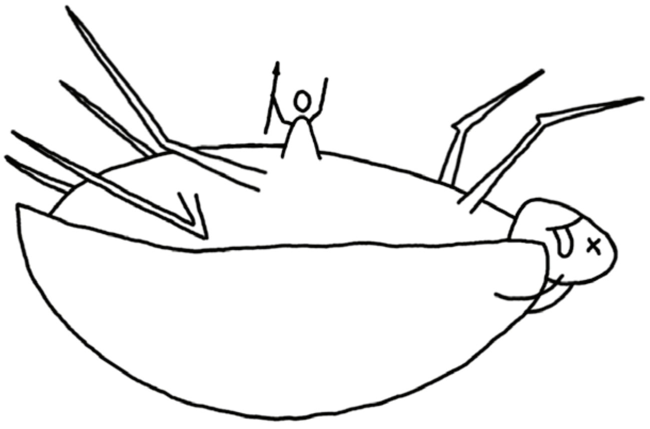
You will, of course, still find and fix previously undiscovered bugs during the stabilisation phase, and such bug fixes still have the potential to introduce show-stoppers. However, with the Zero Bug Policy, this risk is much lower, because you are spending your time proving that the product is stable rather than making it stable in the first place.

HAPPY ENDING?

We were able to restart the Zero Bug Policy in an improved version after the first setback and never had any regrets. However, for us, the Zero Bug Policy represented – to a certain extent – a fight against symptoms of more deeply rooted problems. Drastically increasing the number of teams to prevent delays was a questionable course of action from the beginning. Separating the testing and the development did not help, either.

Nevertheless, I am excited about the Zero Bug Policy and the effects it has had on us. It is simple, effective and inexpensive to introduce at the beginning of a new development. If you already have hundreds of bugs, it gets tougher, but it's not impossible, as our experience has shown.

It may look hard…



…but it's worth it!

AUTOR

Adrian Krummenacher, HTL Electronic Engineer with a specialisation in IT, has been working for bbv Software Services AG since 2000. During this period, he has supported customers in the development of software systems in the fields of robotics, image recognition, accounting and medical diagnostics. Adrian is a dedicated user of agile product development and he has been supporting agile teams as Scrum Master for over 10 years.

REFERENCES

[1] Davison, J. (Producer), & Marshal A. (Producer), & Verhoeven, P. (1997) Starship Troopers [Motion Picture]. United States: Columbia TriStar Motion, Buena Vista International

About bbv

bbv Software Services is a Swiss software and consultancy firm that supports its customers in fulfilling their visions and projects. We develop tailored software solutions and assist our customers with in-depth consultancy, first-class software engineering and solid domain experience to reach a successful solution.

MAKING VISIONS WORK.