Bugs: Management Strategies

By Allan Kelly, August 2014 Software Strategy Ltd., http://www.softwarestrategy.co.uk

Please send comments to xanpan@allankelly.net

Prolog

The following discussion of management strategies for bugs is a draft chapter for the forthcoming book "Xanpan 2: Management Heuristics and Organization" by Allan Kelly. Please see the postscript for more details of this book and where to buy "Xanpan volume 1: Team Centric Agile Software Development."

Please consider this draft as a discussion paper. The author would appreciate any feedback and suggestions for improvement for this chapter.

Bugs: management strategies

Let me ask a philosophical question: *When is a bug is a bug?*

Consider the following scenarios:

- If a developer spots a flaw with their code before they hit compile is it a bug?
- If they hit compile and the compiler flags a syntax error is it a bug?
- If a fault is found during developer testing is it a bug? And if it is fixed is it still a bug?
- If it gets into the source code control system but is fixed before anyone else sees, specifically before a tester sees it, (e.g. the developer realises they made a mistake or another developer see it a fix is made) is it a bug?
- If it is found by a professional tester but never released to a customer is it a bug?
- If a customer reports it is it a bug?

One could probably come up with some more fine grained questions still, and the upper end of these questions could spin out indefinitely but this set will do. Most people would, I expect, answer *No* to the first few of those questions and *Yes* to the last. But those answers are by no means universal and there is a large grey area in the middle.

There are those who believe that almost any developer error constitutes a bug which should be logged and tracked. While this might have made sense in the days when processor cycles were expensive - and most likely programs were written on punch cards - in an age when processor cycles are almost too cheap to measure the most efficient way of finding many minor bugs is use the machine.

Few would argue that if a customer reports a bug it is a bug. But many a customer report has been categorised as a *change request* or *enhancement request* rather than a bug. Which highlights another aspect of the philosophical debate: *When is a bug a change request?* To my mind this is another philosophical question but for many organizations it is actually a serious contractual question which could end in court.

Rather than continue this debate - which is also touched on in the Xanpan Volume 1 Quality Appendix - let me make two assumptions which I hope you can agree with:

- The term bug implies a defect of some sort, and whatever the type of defect rework is required to address the issue. Rework is disruptive and costly.
- The debate about what constitutes a bug tends to be more prevalent when quality levels are low and bug numbers are measured in the tens, hundreds or thousands. When numbers of bugs are counted in single figures, or even a few dozen it is easier to treat all request as work - whether it be bug, change request, enhancement or anything else..

This then is the state we wish to aim for. The rest of this chapter discusses management strategies for dealing with bugs - or to give them a more meaningful name: defects.

Some of these strategies are mutually exclusive but on the whole they may be used together.

But before discussing strategies it is necessary to take a small detour and consider the characteristics of bugs.

Characteristics of bugs

There are two characteristics of bugs which need to be considered when devising strategies for dealing with bugs.

The first characteristic is that, the effort to fix bugs is highly variable. By their nature the effort required to fix bugs is more variable than new work. Nor is it usually possible to break down a bug into pieces and fix each piece separately.

Second, compared to new code it is harder for individuals and teams to accurately estimate how long it will take to fix a bug in advance. Accurate estimation of new code can be hard enough, bugs are even more difficult to estimate. In part this is because as we have just discussed bug fixes are more variable.

Some bugs are hard to identify while easy to fix; others are easy to identify but hard to fix. And new tests must be put in place to ensure the fix works and the bug never returns. On occasions multiple bugs may interact further

complicating matters. And it is even know for bugs to cancel each other out (see *Two wrongs can make a right* (Henney 2010)).

While many, perhaps most, bugs constitute a well defined piece of work which may be done in isolation some bugs are less easy to define. For example, in multi-threaded systems the interplay of almost parallel threads may make it very difficult to define what the problem is.

- While Reuters I spent weeks tracking down a subtle bug which caused options prices to be incorrectly displayed. Even seeing the bug happen was hard: it only appeared in the afternoon, particularly Friday afternoons. At first the only way to see the problem was to place a Reuters terminal next to a Bloomberg terminal, and the only people who did that was Goldman Sachs. When I eventually found the bug it was not in the Reuters system at all but in the Liffe Connect trading system and only occurred when a trader went home early.
- I remember working with one senior developer in California who spent weeks tracking down a subtle multithread book in a Corba object broker. I can still see his drained face after all-night debugging sessions.

Combined these two factors mean it might be impossible to estimate how long it will take to fix a bug, let alone estimate accurately!

The net result is that estimating the effort required to fix a bug is often of marginal value. However when bug fixing work is integrated with other, more predictable, less variable, work the net result is to reduce overall predictability of all work.

Strategy #1: Prevent at source

The first strategy for dealing with bugs is simply to not let them come into existence in the first place: prevention better than cure.

As discussed in the Xanpan volume 1 teams need to seek to minimise the amount of defects and rework required. This is why Xanpan explicitly includes technical practices to prevent errors, many of which originated in Extreme Programming (Beck 2000):

- Test Driven Development whether this be called TDD, Test First Driven Development, Design Driven Development, Automated Developer Unit Testing or another name the essential idea is to write code to test code.

- Acceptance Test Driven Development (ATDD): variations on ATDD may be found under names such as Behaviour Driven Development (BDD), Specification by Example, Automated System Testing and probably some other labels too. All forms represent a step up from TDD.
- Pair programming: two programmers working at one screen, one keyboard, talking and sharing as they co-develop the code.
- Mob or Posse programming: multiple developers working together, with one keyboard and a projector screen, possibly "peeling off" as the day goes on to work on specific sub-sections.
- Code review: formal (Code walkthroughs or Fagan inspections) or informal (asking for a review after a stand-up meeting, tapping a fellow developer on the shoulder and saying "can you please review this?").
- Static analysis tools: frequently hooked into build systems and performing a form of code review.
- Continuous integration (to a source code control system) testing (i.e. regularly executing the tests mentioned above).

This list could go on and newer techniques continue to appear. In general unless I see evidence of teams engaging in some or all of these practices I am sceptical as to whether they are really "Agile." Specifically teams who are not practising test driven development (or a related form of BDD) tend to be guilt until proven innocent in my mind, i.e. I consider they are not really agile although I am prepared to be proved wrong.

Maybe this is unfair of me. I know TDD is not applicable in every case and I know a team might find a better way of working in their technology and environment. But if TDD is not attempted, or dismissed too lightly, I am prone to cynicism.

Strategy #2: Fix close to origin

When a defect comes to light it should be kept within the team and within the current iteration if at all possible and dealt with immediately. The longer it is left to fester the more disruptive a fix will be - because some other code may have been built on top of it making the fix more difficult and increasing the possibly of new defect injection. And the longer it is left the less the

original developer will remember about the circumstances in which it was created.

If a bug is found within an iteration it should be fixed in the same iteration if at all possible. Certainly the feature the bug is found in should not be considered done and should not be released. If it is found too late in the iteration to be fixed it should simply be carried to the next iteration and when fixed the whole feature may be considered done.

The aim should be to fix the bug as close to source as possible so as to limit the disruption it causes. This also means that if a separate bug fixing team (see below) is being used they should not be used to fix bugs created and found in the iteration. Only bugs which escape the iteration should be directed to the bug fixing team.

When a bug is fixed close to source - both in time and people - it reduces the possibility of the same mistake being made again. If it takes six weeks for a developer to fix a bug after creating it then there are six weeks when the same mistake may be made. If the issue is found and fixed in six hours there is far less possibility of the same mistake being repeated, and far less work that should be retested.

If the team are unable, or unwilling, to fix the bug then the whole feature should be deprioritised and removed from the code base. If the business representatives are prepared to ship the feature without a fix then the bug is not really a bug. It can be added to the backlog as another piece of work that will be prioritised on its own merits.

Key to adopting this approach is holding back functionality and features if faults are found. This in turn means the team must organise itself, and its tool chain (e.g. source code control and build systems) and design (e.g. feature toggles) so that features can be "pulled" from releases or never integrated in the first place.

How a team does this will depend on may factors, not least the tools they are using and their source code branching strategy. Some teams adhere to a "one true branch" or "develop on the trunk" strategy which can make pulling a feature difficult although not impossible.

Personally I am not a great believer in single branch development. Although I have suffered "merge hell" and cursed the need to merge branches I believe that branching and merging can be an effective strategy on occasions. To my mind small, focused, short lived (e.g. hours or days) branches - which some would call feature branches - are a very different proposition to long lived (e.g. weeks or months) branches with broad changes across a code base. I also

believe that in modern code control systems (e.g. git and baazar) the very concept of a branch has changed.

In whatever way a team decide to organise their tools and design the key to this strategy is:

> It must to be possible to sideline features which contain bugs written and found in the iteration so that other functionality can be released and used.

Strategy #3: Quick decision, quick fix

If for any reason the a bug escapes an iteration it needs to be fixed quickly. As mentioned before, the longer a bug lives the more difficult a fix becomes.

Saying "fix it quick" implies that a quick decision is made to fix the bug once it has escaped. This also means that the opposite decision is also open: no fix. It is entirely permissible for those in authority - product owner, manager or someone else - to decide a bug will not be fixed. As long as the organisation and team are prepared to live with this situation and the bug is marked "Closed - will not be fixed."

(This of course reopens the philosophical question of "what is a bug?" but we will not reopen that discussion at this point.)

The aim of implementing this strategy is to limit the number of bugs a team must live with. The number of bugs which need to be administered and managed, when bugs are rampant it becomes very difficult to actually reason about bugs and what should be done. While bugs are open they tend to soak up time, energy and morale.

Yes: it is best to fix any bug shortly after it is identified. But there is little value is keeping a trivial bug open for months or years, long past the point when anyone expects it to be fixed. Such bugs get in the way of really being able to manage bugs and quality.

Making a quick decision requires the authority to make the decision is available to make the decision. That probably means devolving authority to those closest to the work; and those with authority need to be available to make the decision. There is no point in putting this authority with a manager who can only spend a few hours a week with the team.

Strategy #4: Active bug management

In an organisation with many bugs - which implies just about any organisation that has an electronic bug tracking system - there is much that can be done to manage bugs away. Normally the sheer number of bugs open makes it difficult to see which bugs are serious and which trivial, which need to be fixed and which should be simply closed "do not fix." Not only this but masses of bugs are morale sapping and waste management time.

Of course managing bugs away does little in itself to improve the software quality but it does make managing a lot easier. When bug lists are short and under control work can be directed to where it is needed. When bug lists are long and out of control - all too common when an electronic tracking system is used - the sheer volume of bugs overwhelms effective management.

In a sense any bug list is just another backlog of work to do, a *Bug Backlog*. And for a team that backlog is just one source of *work to do*. The bug backlog can take its place alongside the *Product Backlog* (encompassing the Opportunity and Validated backlog where that model is used), the *Technical Debt Backlog* and any other backlog that is significant enough to get its own name.

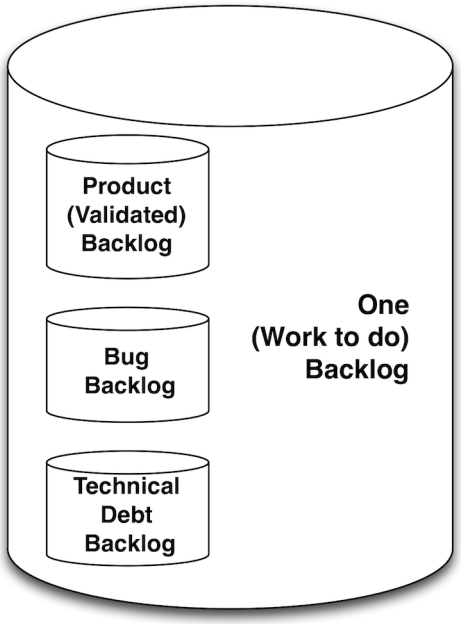


Figure 1: Ultimately all backlogs are one backlog - work to do

Cumulatively these are all one backlog, they are all work to do for the team. Segmenting can be useful for analysis and reasoning but it does nothing to increase the resources available for fixing them.

The team have the capacity they have: there is no bug fixing fairy who will do it for them. They should work to increase the capacity but there will always be choices about whether to undertake rework on a defect or to do new work.

Managing bugs away demands that management devote time to bugs. I suggest this takes the form of a weekly meeting of a fixed duration, e.g. one or two hours. The meeting includes:

- A representative from the development team, usually a development manager, who has knowledge of the system - although not necessarily technical knowledge - and the authority to have work done.
- One or more of the developers who will do the work: these will add a technical voice to the discussions.
- Representatives from customer services: speaking for real users and/or customer.
- Representatives from Testing/Quality assurance (if the organization has such group).
- Sometimes more senior managers will join the meeting if bugs have a high profile with customers.

Naturally a small meeting is better than a large meeting and the above list is dangerously close to being too long. The aim of listing so many roles is to ensure the meeting is properly able to reach a decision. There is no point in the meeting coming to a conclusion - say to fix a bug or change priority - only for, say, the Test Manager to turn around the next day and say "I don't agree, change it back."

The meeting occurs in the same timeslot each week - using the principle of rhythm outlined in volume 1. The meeting splits into two parts: the first part concerns itself with new issues and the highest priority bugs. This part repeats week after week.

The second part of the meeting also repeats week after week but each time it starts where the previous meeting left off. It reviews every single bug logged and open.

The first item on the agenda is to review all bugs reported since the last seeking. Most likely test or customer services will have assigned a priority to the bug. If it was particularly serious, say effecting a live customer, work may already have been started, and even completed, on a fix. So the first action is to review all the new priority decision made in the last week and confirm the or change them.

Many, if not most, companies use a priority scoring system form 1 to 4 (or 5) to rank bugs. Priority ones, P1s, are the highest priority while P5s are normally trivial.

The next meeting action is to review all the top priority bugs, normally those designated P1s. Having done this the absolute priority of these bugs needs to be decided. That is, which of the P1s should be fixed first? Which second? And so on all the way down to the last P1.

This might be a time consuming and painful exercise, especially the first time it is done, but it should be so. Bugs are painful and if people do not personally feel the pain the motivation for fixing them will be diminished. Besides: customers and users are feeling the pain so why shouldn't some of those responsible for creating them feel some pain? Empathy has a role.

In reality after this has been done a couple of times, and once the development team have settled into a pattern and know approximately how many bugs they fix each week the exercise becomes less time consuming and less painful.

I have been known to write all P1s on red index cards (well pink cards to be exact) and have the meeting physically order the cards. This can be a very powerful technique for illuminating priorities and the pain customers have.

If there are too many P1s to make this process practical it is a clear sign that something is wrong. Either there are too many bugs classed as P1, and some should be moved to P2, or the company has a really serious quality problem and urgently needs to devote more time and energy to resolving the situation.

With P1s dealt with the meeting moves to the other bugs. The remaining time is used to review all other bugs. This process is likely to take many meetings over several weeks, each meeting does as much review as it can in the remaining time. The next meeting picks up where the last one left off.

Starting with the P2s the meeting should review the bug and decide:

- Is it still relevant? Maybe something has changed in the system and this problem is no longer seen.
- Does it has the correct priority, should it be higher or lower?

- Whether it is a duplicate of something else or can be closed for some other reason (e.g. the customer who reported it has gone bankrupt.)

When time runs out for the meeting someone records the position in the bug list. The meeting finishes until the following week. The next meeting follows the same agenda - review weekly decisions and prioritises P1s - before continuing to the bug review where it left off.

Once all P2s have been reviewed the meeting advances to P3s and so on. Once the whole list has been reviewed the oldest, lowest priority bugs, should, on mass, be marked as closed.

For example, if after reviewing every bug over a six week period the company has ten P5 "trivial" bugs which have been open for more than one year they will be closed. Such bugs are unlikely to ever be fixed and have failed to demonstrate their importance.

Next time the meeting convenes and after dealing with the P1s will start afresh with the P2s and traverse down the whole bug list a second time. The aim is to remove all the duplicates, irrelevant, and other clutter bugs and ensure consistent and meaningful prioritisation.

At the end of the second pass the close criteria is tightened. For example, all P5 bugs which are more than six months old are marked as "Closed - will not fix."

After several passes - in many meetings - the list will be shorter, the team will have a better idea of what bugs they really face and the meeting can cease the reviewing all the lower priority bugs. In time they may repeat the exercise but after several passes the true state of affairs will be a lot clearer.

Strategy #5: Bug fixing sub-team

This strategy starts from the decision to ring fence a proportion of the available capacity for bug fixing. This normally takes the form of designating one or more developers as the "Bug Fixing Team". Their work is focused exclusively on fixing bugs.

By ring fencing one or more people as bug fixers the team devotes a proportion of their capacity to fixing bugs.

This approach also makes scheduling other work more predictable. As previously mentioned bug fixing is far more variable and difficult to estimate than other types of work. Thus overall predictability of a team suffers. When

this variability is separated and managed as a separate stream of work the predictability of non-bug fixing work improves.

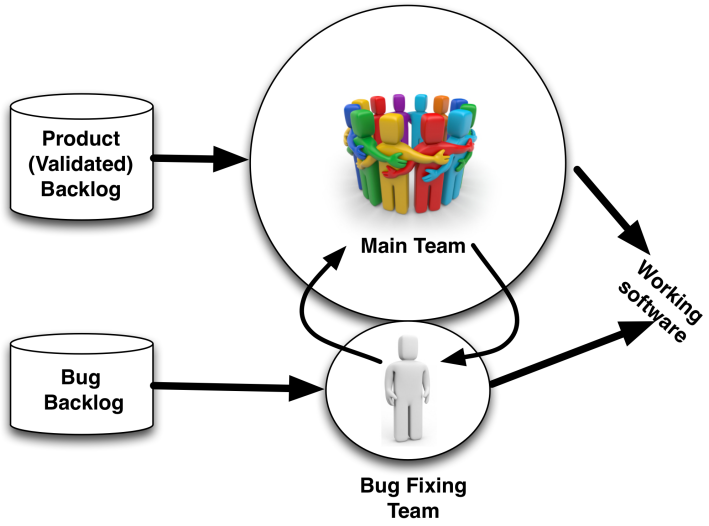


Figure 2: Main team focus on new product work while bug fixing team work on bugs

To be clear: bugs made recently - within the iteration or within the last few iterations - should most definitely be directed back to the main development team, as per strategy #2. Their work is not done if it contains bug.

There should be no incentives for doing a *quick-and-dirty* piece of work which someone else needs to tidy up. Not only does this result in poor quality code but it also obsfiscates any attempts to apportion costs correctly.

Developers should be rotated from the main team into the bug fix team on a regular schedule. For a team of four this might mean that every iteration one person fixes bug, the next iteration they rotate back to the main team and someone else fixes bugs. Over the period of four iterations everyone will spend one iteration on fixing.

The rotation then repeats. So over a period of eight iterations everyone spends two iterations fixing bugs.

These two sub-teams sit together, indeed there is no need for anyone to move desks at all. People sit in their regular places and continue to share knowledge and assist one another. If this iterations fixer needs help from someone else they asks.

While the designated bug fixer will continue to attend iteration planning meetings, stand-ups and so on they bug fixing is effectively run as a separate iteration with its own schedule of work - driven from the bug prioritisation meeting described in strategy #4.

In fact since the time is ring fenced and the developer will work on one bug at a time - the highest priority bug first, then the next and so on - they may well abandon estimation altogether. In this context estimation adds little and since it is highly variable may actually cause confusion.

Can we also do small changes in the fixing team?

A question that comes up a lot. The short answer is: *you can do anything you want, but does it make sense?*

Having the bug fixing team do small changes might make sense when there are few bugs that need fixing but in that case it probably makes more sense to fold the sub-team back into the main team and do the work there. (Strategy #6.)

When there are a lot of bug then using the fix team to do small changes will impact their ability to do bug fixes and stay on top of bugs. Using them as such destroys the effectiveness as a bug fix team.

So while one can have the bug fix team work on non-bug work it defeats the purpose of having a dedicated fix team.

That said, if there are few bugs to fix, it might make sense to have a team addressing "small" pieces of work some of which happen to be bugs.

Strategy #6: Fix within team

The alternative approach is to have one development team work on new work and bugs with no ring fencing.

Having the main development team fix bugs as part of their regular work can be an effective strategy. The strategy tends to work best when few new bugs are being uncovered and fixes requested.

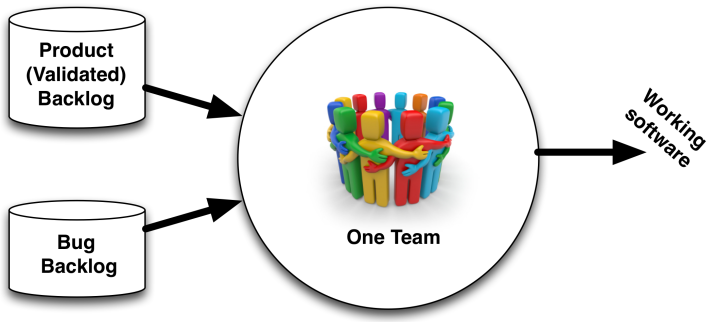


Figure 3: One team fixing bugs and new product work

Under this approach the bug list - or bug backlog - and backlog of work to do - whether called product or validated backlog - is, as already described, one long list of *work to do*. The Product Owner needs to consider both lists when choosing which work they would like done in the next iteration. Hence they need to consider the value of a bug fix against the value of any other piece of development work.

As with any other piece of work the team accept the bug fix as a piece of work to be done and endeavour to fix it. Given the greater variance in the nature of bug fixes any piece of work prioritised below the bug has a lower probability of being done.

(It is worth noting that if there are a lot of bugs and the main team spend most of their time working on bugs then the main team is in effect a bug fixing team. So another strategy, seldom seen, is to have a large bug fixing team and a small new development team.)

There are several advantage to having the main team fix bugs. Firstly it completely removes the possibility of any incentive for individuals to short-cut work by thinking that someone else will patch up the result.

Secondly it removes the need to designate anyone as "bug fixer" or to devise a rotation schedule.

Thirdly it increases chances that the most appropriate person to fix a bug will get to fix the bug.

However there are, besides reduced predictability, several disadvantages. More time will be spent in discussions about whether a bug should be fixed or not, and what the relative priority is vis-à-vis other pieces of work.

Experience shows bug fixes are less attractive to both sides - technical and business - then new work. Consequently new work is schedule more often and

bugs less often, bug lists grow and become more difficult to manage. And customers never get fixes.

To estimate or not to estimate?

When a ring-fenced bug fixer or bug-fix team is being used it makes little sense to estimate the effort required to fix a bug. Bugs should be prioritised according to business need rather than development effort. The team has decided how much effort will be put into bug fixing, i.e. the effort of the ring fenced fixer(s).

In this scenario the time taken to estimate a bug is time that could have been spent trying to fix it. Since bugs fix estimate are unreliable the information content of such an estimate is very low so cost-benefit analysis is not going to be reliable.

When the main team is fixing bugs as part of their general work load it makes more sense to estimate the effort required to fix a bug so that work can be scheduled in the same fashion as other work. However given the higher variance in both bug estimates and time to fix this will result in a more variable velocity - as described below. Consequently future iteration will be less predictable because the velocity data will be more variable.

Indeed, since future iterations will themselves contain a mix of bug fixes and new work they will themselves be more variable. The smaller the team the greater this effect because smaller teams find it more difficult to absorb variability.

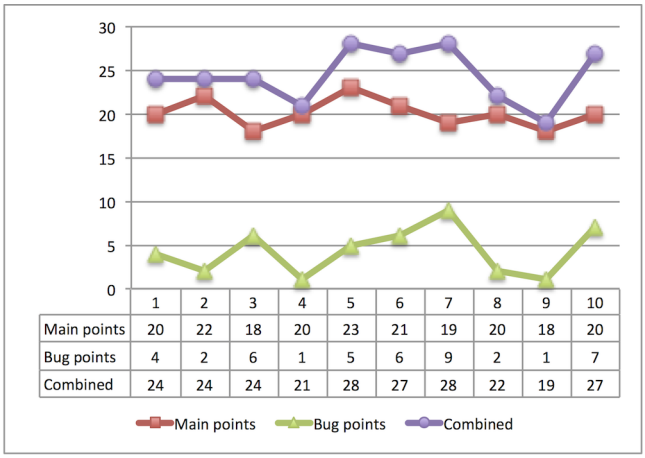
Variability may be reduced if bugs are not estimated. Instead the team might aim to fix a set number of bugs per iteration, e.g. three. While some bugs will take more time and some less and velocity will suffer the overall variability will be less and therefore the predictability will be greater.

With enough data average time taken to fix a bug might stabilise. Iteration to iteration predictably will be less but in the longer term (e.g. next quarter) prediction should be more reliable.

Variability?

Consider the example shown in the next graph. When bugs are estimated in the team and treated as any other work - shown by the combined purple "combined" line - the team averages 24.4 points per iteration with a standard deviation of almost 3.1.

Buy if bugs are separated out - the green line - the average score is 4.2 with a standard deviation of 2.75 while the non-bug fixing work averages 20.1 points and a standard divination of 1.6. In other words, when bugs are excluded from the teams velocity score the score is more stable and is a more reliable predictor.



	1	2	3	4	5	6	7	8	9	10
Main points	20	22	18	20	23	21	19	20	18	20
Bug points	4	2	6	1	5	6	9	2	1	7
Combined	24	24	24	21	28	27	28	22	19	27

Figure 4: Variability with and without bugs

(This example has assumed a standard distribution for bug fix and other work effort. This is almost certainly not the case, you need to do your own analysis. The assumption is made here to illustrate how disruptive bundling two different types of work together can be.)

Another option

Another option might be to allow a fix amount of effort on each bug, say one day. Each fixer starts the day working on one bug, if it is fixed by the end of the day good, if not then fix attempts are suspended and the next day the fixer moves to the next highest priority bug. The data gained in the one day of (unsuccessful) effort can be discussed in the next bug review meeting.

This option is somewhat theoretical as I have never seen it actually done. It makes logical sense but would require some discipline to implement.

It does beg the question: *what if the developer finishes part way through the day?*

The obvious option is to start on the next bug immediately. However this would require them to set a deadline 24 hours hence and respect it.

Another option is for the developer to go home early. Attractive and elegant as this may be it is hard to imagine many organizations endorsing this approach.

A half-way house might be to let the fixer start on the next bug but leave the deadline in the same place. For example: if Dave starts work on bug 1234 at 9am on Tuesday and finishes it at 2pm the same day he would then start on bug 1235. The time-out on this bug would be end-of-day Wednesday as it would have been had he started it on Wednesday morning. In effect he gets an early start on 1235.

Choosing between strategies

Deciding between these strategies is not as difficult as it might seem. Four of the strategies are always valid:

- Strategy 1: Prevention
- Strategy 2: Keep close
- Strategy 3: Quick decision
- Strategy 4: Active bug management

When several passes of the bug backlog have been completed and strategies 1 and 2 are embedded than active bug management may be reduced. Until then it is usually a good idea.

The real choice comes between the last two strategies:

- Strategy 4: Bug fixing sub-team
- Strategy 5: Fix within team

As a guide, fix bugs within the main team when:

- The number of bugs is low
- The need to fix bugs is limited
- Predictability is not of primary importance

When one or more of these conditions hold then it is best to fix bugs in the main team. In the longer term, a stable team working to improve quality and a track record of delivering will probably be best off adopting this strategy.

But bug fixing teams are preferable when one or both of these conditions hold.

Conversely, use a bug-fixing team when:

- The number of bugs is high: definitely employ active management of bugs (strategy #4) and use a ring-fenced sub-team to address bugs.
- Predictability is very important

These heuristic are summarised in the diagram below.

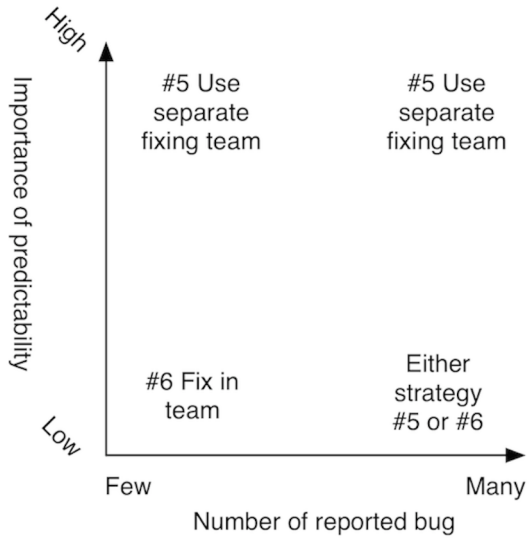


Figure 5: Summary of bug fixing strategy decision

Under pressure to hasten deliveries, and improve predictability teams may be tempted to forego bug fixing altogether to drastically throttle fixes in the main team. This is almost certainly a mistake. Unfixed bugs represent a major risk to predictability and will complicate new work, which in turn will increase the time.

Given these heuristics the decision may seem easy but these are heuristics, there will always be other forces at work. Indeed these heuristics overlook another important force: team size.

Team size

Small development teams will always struggle to deliver new functionality when they must also handle large numbers of bugs. In these cases Active Bug Management becomes a vital tool.

The problem for a small team is that devoting any resource to bug fixing has a significant hit on delivering new functionality. For example a team of three developers which ring-fences one developer would devote 33% of their capacity to bug fixing while a team of six developers devoting one person to bug fixing still has over 83% of their capacity for new work. A team of 10 could put two people on bug fixing and still have 80% of their capacity for new work.

Thus it is more difficult for small teams to adopt strategy #5 (bug fixing sub-team) and will find pressure for them to follow #6 (fix in main team) but these are - because of the same maths - exactly the same teams which suffer the most schedule disruption from other sources. Small teams struggle to absorb variability whether from bug fixing or other sources.

When predicability is important, there are a lot of bugs which need addressing and the team is small something has to give. If nothing is done predictability and quality will suffer, the team may keep up appearance of being on schedule but quality will be cut and this will eventually undermine the illusion of schedule.

I have seen, even been complicit in helping, teams hide such problems. If the organization can recognise the problem and steel itself to act there are two options:

- Option A is to simply accept the unpredictability or take no action to it. This might not the most politically acceptable cause of action but it is the default and possibly the most common.
- Option B is to take action. The question now is what action?

Bug fixing could be forgone but this too is risky, ignoring bugs may simply push the risk further down the line and increase the impact. The bugs that cannot be ignored will reduce predictability and slow the team down.

Giving the team more time to deliver might work but the longer the time to delivery the more the requirements will change.

Giving the team less time to deliver, moving to an immediate delivery should at least establish a safe harbour. The initial product may not contain all the

desired functionality but it would at least show truthful where the team are at. Continuing to make small incremental additions and fixes with regular releases will allow the team to proceed safely if somewhat slowly. Overall they may take longer but at least they have a fighting chance.

Increasing the team size would address problems directly. Although due to Brooks Law the team would still require more time.

A larger team could absorb variability better and open the option of ring fencing fixing capacity (strategy #5). Although it would take time for predictability to stabilise as new staff become familiar with the product and the code.

If increasing the team size is not possible, or would take too long to improve the situation the final option is to stop the work. When all viable options are precluded continuing the work as is - small team, lots of bugs and demands for predictability - the work is very high risk. The team is unlikely to succeed so it probably makes sense to make this decision early rather than let the story play out.

In such a situation it may be better to redirect all resources to a less challenged endeavour.

Other strategies

There are two other fix fixing strategies that are sometimes seen although neither are normally recommended. Both strategies are attempts to compromise between having a fixing team and working in the main team, as is often the case with compromises they less than optimal.

Stop and fix "bug blitz"

Team using this strategy stop all new development work and instead just fix bugs. In effect this is what many traditional teams do when they have finished the new development work. One problem with this approach is knowing when to stop. If the objective is to just fix bugs shouldn't the team fix bugs until they have fixed a set number? Or when the total number of bugs has been reduced to some "acceptable" level?

In either case the question arises: what happens when the number of bugs rises? Do the team stop and blitz again?

This points to the second and more significant problem with this approach: when bug blitzing becomes standard practice it detracts from regular efforts to keep quality high and bugs under control. Why bother with strategies 1, 2, 3 and 4 if all the bugs are doing to be batched up till the end?

Since bugs are being built up the team moves away from the position of having a always - or at least regularly - releasable product. Each burst of new work needs a subsequent bug blitz. This in turn produces a culture of "do new work" and creates an incentive to postpone bug fixing.

Business representatives are usually loathed to sanction a bug blitz because it implies new, valuable, work is not being done. A reinforcing downward circle of poor quality is created.

There are a couple of exceptions when a bug blitz might be appropriate. Many teams use a blitz as a buffer between one project and another.

This is not uncommon in corporate environments where teams are prevented from starting on new work until some ceremony has occurred, a formal sign-off or a financial start-of-year being the most common. Teams which are not allowed to work on CapEx Project work fix bugs while they wait.

One would not really call this state of affairs "Agile" but it can be a useful way to spend otherwise lost time.

The second occasion a bug blitz can make sense is as a one-off exercise to transition from the old way or working to the new. A team might decide to spend several weeks on bugs. During this time an intensive bug review (strategy #4) would be undertaken in parallel with a fix and test exercise.

At the end of the period the team would push out a new release and switch to strategy #5 or #6 for future bugs.

Day a week

Strategy #5 described ring-fencing capacity by separating one or more programmers and having them work for the entire iteration on bugs. So over a two week period in a team of 5 developers would spend 10 days bug fixing and 40 on new work.

Another way of formulating this ration would be to have everyone fix bugs on a Thursday. This would see 5 people devote two days to bug fixing (10 days again) with the other days used for new work.

This looks attractive because it shares the load and avoids one person being the fixer. However this is less clean cut than strategy #5 because it means

everyone must once week switch their minds from new work to bugs and then back again. Partially done work is left hanging and then, perhaps, partially fixed bugs are left in hanging.

One might minimise this effect by making Friday (or Monday) the fixing day. However this might create other issues.

The two biggest arguments against this approach are firstly: what happens to a bug which isn't fixed on the day? Does it interrupt routine work the next day or is it left until the next week?

Secondly in time the business representatives may well objective to seeing the entire team switch away from new - money earning - work to bug fixing each week.

Again this might be an effective strategy in the short term when combined with Active Bug Management and Quality improvements but in the longer term is likely to be difficult to implement effectively.

Why not?

Before leaving the subject of bugs it is worth considering why so few teams choose to pursue these strategies - or any other for managing bugs.

The answer to this question may be as simple as misplaced optimism. The desire for bug free, or near bug free, software, the desire for things "to be different this time" is so great we allow ourselves to be wish away the problem - or at least deny its existence.

However I believe there are several reasons more significant reasons why companies in general fail to devote appropriate resources to bugs and take steps to deal with the problem. And I believe that naming and explaining the reasons is a first step towards changing everyones mindset.

Quality is seen as negotiable

Firstly too many managers have come to believe quality is a negotiable property which can be traded-off against other properties. Specifically quality is seen as a parameter in determining schedule: higher quality leads to a longer schedules while lower quality leads to shorter schedules and faster deliveries. While it may be true that not fixing bugs is faster than fixing bugs this does not hold until the bug is created. If bugs are prevented then overall development schedules will be shorter.

It is as if there is a dial on the wall marketed quality. Too many people believe that turning the dial down, to low quality, will increase the speed of the development team. Conversely turning the dial up, to higher quality, will slow the team down.

Figure 6: False belief that turning quality down increases speed

This might be true in some industries but it is not true in software development. The work of Capers Jones (C. Jones 2008) shows clearly that lower defect potentials and higher defect removal efficiency lead to shorter schedules. Jones is not alone in this, other authors have found the same result. Indeed IBM first published such a study in 1974. (See Jones again for details.)

This finding should not be surprising. Philip Crosby founded an entire quality movement with his book *Quality is Free* (Crosby 1980). Such findings have been found in silicon chip manufacturing, car production, rocket manufacturing and many others. High quality is the foundation of many Lean principles and practices (Womack, Jones, and Roos 1991).

While quality might be free in the long-term it is probably more correct to say, as Niels Malotaux suggests: "Quality is free but only if you invest in it." In other words: retro-fitting quality (i.e. bug fixing) is expensive but expenditure to prevent and remove bugs at the earliest opportunity will pay back many times over.

Figure 7: In software turning quality up increases speed

Cost of a bug

At first sight a bug is cheap: "We can save a bit of money by taking a risk, we can accept a bit of shoddy code in the same way a window clearer skips the odd pane here and there to go faster. If anyone finds a problem it won't take very long to fix it and issue a patch."

One might get away with this thinking in an early stage start-up with few customers but as customers number increase the costs of riding a bug increases and the costs of fixing tend to increase massively. Lets stop and think of the costs associated with a bug:

- The cost of writing the code in the first place (Development cost).
- The cost of testing it (Test or development cost).
- The cost of false positive bug reports (Test cost). When bugs are common so too are false positives, people jump to assumptions about what doesn't work more quickly.
- The costs of reporting the bug (Test cost).
- The cost of arranging a fix (Management time).
- The cost of fixing (Development time).

- Cost of fixing new bugs introduced by the fix: it has been suggested (C. Jones 2008) that 7% of bug fixes inject a new bug (Test and development time).
- The costs of support desk calls when a customer finds a bug (Support desk time).
- The costs of repeated support desk calls when multiple customers find a bug (Support desk time).
- Cost of administering duplicates bugs: entering, identifying, merging, etc.
- The costs of helping customers who are struggling to perform a task which is afflicted by a bug (Support desk time).
- The costs of issuing bug notification or "work around" notes to help effected customer (Support desk time).
- The potential lose of customer time, money and even business opportunities (Customer costs).
- Cost of shipping software late.
- Cost of poor predictability: costs incurred because deadlines are missed or other resources deployed at the wrong time.
- Cost of additional countermeasures to reduce risk.
- Cost other work not done.

Many of these costs occur for false positives - reported bugs which are later shown not to be bugs. And many of the costs occur each time a duplicate bug is raised.

This is a long list and we have yet to consider the cost of actually fixing the bug, retesting, issuing the fix, updating release notes, manuals, support desk databases and informing customers. Nor have we considered the cost of managing customer expectations and potentially lost sales.

Perhaps dwarfing even these costs is the potential loss of revenue to a company using the software which contains the bug. This might be direct (e.g. the bug stops the client performing some revenue generating action) or it might be indirect (e.g. loss of reputation.)

One of my clients produces a system used to position oil rigs correctly. I once asked how much it would cost to move an oil rig if it was incorrectly positioned, by say 1 meter or such: "of the top of my head, say $25 million" was the answer.

As discussed in the Quality Appendix to Xanpan volume 1 the need for high quality, few bugs, should not be taken as a reason to gold plate and over engineer software system. Such over engineering efforts create problems of their own. However there is an exceedingly good case for attempting to minimise the amount of rework, or work arising from work which we thought "done."

Value of a bug

Just as it is wrong to look only at the cost of new development work it is wrong to only look at the cost - whether in money or time - of fixing a bug. In both cases it is important, probably more important, to consider the value the work will bring. In both cases assessing value can be difficult. In the case of bugs there are additional points that may need considering to assess the value of a fix.

In the case of bugs the financial business value of a fix may be very low but the reputation value an be very high. For example, a bug might not prevent the software being used for its primary purpose but the need on a regular basis for users to work around problems may do damage to the image of the creators of the product.

Sometimes the value of a bug fix might be necessary to full fill contractual obligations. How much of the contract can be attributed to the bug might be debatable but if the bug prevent final acceptance and payment then it has a sever effect on cash-flow.

Then there are the technical benefits from fixing bugs. Bugs which regularly disrupt service or hinder programmers and testers from doing their development jobs might merit fixing even if the business customer seldom sees the issue. Similarly problems which lead to large number of support desk calls or interruptions to other work may be valued in terms of internal savings.

CapEx and OpEx

If you have never come across the terms CapEx and OpEx consider yourself lucky. They are accounting terms:

- CapEx is short for "Capital Expenditure": Money spent under this heading appears on the balance sheet as an investment, so $100,000 of spend is balanced by the creation of a $100,000 asset.
- OpEx is short for "Operational Expenditure": These monies do not return, they are gone. $100,000 spent on $100,000 is a $100,000 the company no longer has.

Some companies monitor and manage CapEx and OpEx more than others. The senior managers, their board members or even investors impose limits or set targets on each category.

Complicating things further some CapEx expenditures qualify for tax incentives so $100,000 CapEx invested on software development might entitle the company to a 10% tax rebate of $10,000.

New development work is often considered CapEx while bug fixing is considered OpEx. It is not only in software development that such accounting conventions cause problems. Exactly these conventions contributed to the failure of WorldCom in 2002: the company counted all network maintenance as CapEx rather than the more usual OpEx.

One can argue with these accounting conventions: a bug fixed as part new development on an existing product is CapEx but if fixed as an explicit bug fixing effort then it is OpEx. But these are the conventions that exist. While it is probably a sign that accounting has yet to catch up with the digital age there is little prospect of things changing soon.

These categories, and the management process associated with them can distort priorities and encourage bugs to be left unfixed. Rational strategies, like ring fenced teams, for addressing bugs are sometimes denied.

Finally

Perhaps one of the reasons bugs present so many problems is simply that they should not exist. The mental pull of this view leads us to pay less attention to bugs than other work.

The good news is that teams who try can vastly reduce the number of bugs they have much to gain. The bad news is that few teams will eliminate them entirely.

Indeed as teams get better at avoiding simple coding bugs the energy we put into finding these bugs will find new bugs, bugs relating to desired

functionality, or usability, or some other category. Arguably preventing these bugs is harder but the value in fixing these bugs is also higher.

Indeed many of the things we call "bugs" are temporary learning objects. They allow us to learn more about what is actually wanted. Without herculean efforts it would not be possible to foresee and prevent these. At this point the relationship between higher quality and shorter delivery schedules may break down. However this is a hypothesis and will only be clear in retrospect.

Until then teams should strive to prevent bugs as far as possible and efficiently remove those defects that do sneak through. To use the terminology of Capers Jones: *Strive for low defect potential and high, and rapid, defect removal.*

References

Beck, K. 2000. *Extreme Programming Explained.* Addison-Wesley.

Crosby, P. B. 1980. *Quality is free: the art of marking quality certain.* New American Library.

Henney, K. 2010. *97 Things Every Programmer Should Know, Henney.* O'Reilly.

Jones, C. 2008. *Applied Software Measurement.* McGraw Hill.

Womack, J. P., D. T. Jones, and D. Roos. 1991. *The machine that changed the world.* New York: HaperCollins.

Postscript

This is an draft except from the forthcoming book "Xanpan 2: Management Heuristics and Organization" by Allan Kelly. You can register your interest in this book at https://leanpub.com/xanpan2. The author would appreciate any feedback by way of comments or suggestion for improvement. Please send comments to xanpan@allankelly.net.

Xanpan volume 1: Team Centric Agile Software Development is also already available both in electronic form and printed.

- Electronic: https://leanpub.com/xanpan
- Printed: http://www.lulu.com/shop/allan-kelly/xanpan-team-centric-agile-software-development/paperback/product-21598658.html

About the author

During his career Allan Kelly has held just about every job in the software world, from system admin to development manager by way of programmer and product manager. Today he works helping teams adopt and deepen Agile practices, advising companies of development generally and writing far too much. He specialises in working with software product companies and aligning products and processes with company strategy.

He is the author of three books: "Xanpan - team centric Agile Software Development" (https://leanpub.com/xanpan), "Business Patterns for Software Developers" and "Changing Software Development: Learning to be Agile"; the originator of Retrospective Dialogue Sheets (http://www.dialoguesheets.com), a regular conference speaker and frequent contributor to journals.

More about Allan at http://www.allankelly.net and on Twitter as @allankellynet (http://twitter.com/allankellynet).

Software Strategy Ltd.

http://www.softwarestrategy.co.uk