Issue Types

Stories

A "Story" represents new (or revised) functionality that we intend to ship to the end user. It is the basic building block of our development work. In its initial conception, a story should represent *what* we want the software to do, but not necessarily *how*. It should have an explicit (or obvious) motivation from the end user's perspective as to *why* they would want this. It is called a "story" because it should represent a narrative of the user interacting with our software to accomplish a real-world goal. Certainly the details may draw us into engineering considerations that are far removed from the end user experience, but they should always be traceable back to the end user's "why". As the story evolves from its initial conception, it will accumulate more of the "how" decisions.

A story should represent a coherent package of work, and be expected to take less than 2 weeks' worth of engineering effort. Stories larger than this should be broken down. Ideally, the work should be broken down into a series of stories geared towards achieving a "minimum viable product" as quickly as possible, and then evolving the system towards greater utility in clearly-defined stages. (This approach can be said to split the work into "vertical slices", as opposed to "horizontal slices". Stories should *not* be broken down into horizontal slices, e.g., first do all the under-the-covers work, then do all the middle tier stuff, and then finally the user interface. This is undesirable because it represents long stretches of time where the software is in a poorly-understood and loosely-verified state, making it hard to say what we've accomplished or how much is left to do.) Another technique to handle large efforts over a long timeframe is to hide them behind a Debug option and leave the old subsystem in place until the new subsystem is ready to replace it.

Epics

An "Epic" is a grouping mechanism for a series of related stories. An epic does not represent work in and of itself -- it must be composed of one or more stories. An epic will almost certainly span several sprints, and may even cut across multiple releases of the software. Epics are largely optional, and should only be used to the extent that they are helpful in organizing our work. If a large story is broken down into multiple stories, then almost certainly they should be linked together under an Epic.

Bugs

Bugs are bugs -- software behaviour that deviates from our intention. See "Bugs vs Stories" below for a more detailed explanation of the difference between bugs and stories.

Tasks

A "Task" represents other necessary work, such as infrastructure tasks (e.g., build machine configuration, QA automation, etc.) or ongoing maintenance. A task differs from bugs and stories in that it does *not* represent work product that gets shipped to the customer.

Subtasks

Any issue type (Stories, Bugs, or Tasks) can have Subtasks. This allows you to break down the issue in a more structured fashion. Each subtask shows up separately on the swim lanes, and can move independently through the workflow.

Use of subtasks is optional. Generally, if a story needs to be broken down in a structured fashion, we break it down into multiple stories under an epic. All the particular details are handled in prose in the Description field. Subtasks sort of fill a middle-ground between loosely-structured text in the Description field and the full-on structure of epics/stories.

Bugs vs Stories

To do things well and in good order, we should only write up bugs for things that are really bugs, for a restrictive definition of "bug". All other work should be handled as Stories. For clarification:

Bugs

- When actual behaviour differs from intended behaviour ("intended" meaning: the developer actually attempted to make it behave a certain way)
- Regressions (something that previously worked is now broken)
- Unintended side-effects
- Manifestly obvious, egregious oversights

Not Bugs

- Newly discovered requirements
- Unanticipated scenarios
- Disagreement over how it should behave
- Aesthetic judgements
- Breakage due to external factors (e.g., new OS versions, updated hosts or plugins)

The reason for a restrictive definition of "bug" is that we want to fast-track bugs (especially regressions), while allowing stories to proceed through the usual process in priority-order. Bugs should be exceptional occurrences; we don't want unnecessary fast-tracking to overwhelm the normal flow of development.

Writing Up Issues

Issues are written up in Jira. First determine if it is a Story, a Bug, or a Task. The issue should be given at least a Summary. Include other information (if known) in the Description field. Bugs should have easily-followed reproduction steps, along with affected OS's in the Environment field, and build number. Select a "Component" field, which has the useful effect of automatically assigning it to the responsible engineer. If there are questions about which Component or who should be assigned, assign it to the Scrum Master.   If it's already been decided what release it should be a part of, set the Release and move it to the top of the backlog (otherwise, it will be triaged by the Scrum Master later). If it's a regression, add it to the currently-active sprint.

The Backlog

The backlog is a ranked list of all the work we intend to do, or even are thinking about doing, ever in the future. At this level, it's not tied to any particular release or timeframe. An important aspect of the backlog is that it is strictly ordered, with the most important items on top. The definition of "important" is a little flexible -- it's not just what would be most valuable, but also some consideration of a back-of-the-envelope Return On Investment calculation (business value divided by engineering effort), and some consideration of any overarching plan of what features we want to land when. The stories are rough and high-level at this point.

Releases

Releases are planned releases of software to the market. They can also be interim releases (such as betas), or internal milestones.

For a release, we create a release backlog. we take items from the general backlog and move them into the release backlog. This becomes the definition of the release -- everything that we intend to ship for that release. Certainly, plans change over time, but it should always represent our current best understanding. Like the general backlog, it is also strictly ranked, with the idea that the stuff we really need gets in first, and then the stuff at the bottom gets considered for trade-offs of features-vs-ship-date. As items bubble up to the top of the release backlog, they need to be further defined.

For a release, we may create a "consider" line in the backlog. Issues above the line are expected to be in the release. Issues below the line are being tracked for consideration -- they may be included, depending on how we want to trade off getting more features against the ship date.

Backlog Grooming

The backlog should be “groomed” regularly. This involves triaging new issues (integrating them into the priority order and assigning them a release if known), and ensuring that the priority order and release assignments are up-to-date with our latest thinking. Near the end of the cycle, grooming also involves making the tough decisions as to which issues will make the cut and which issues will get pushed to a later release.

Issues enter the backlog without much detail. This is to be expected. As they move closer to the top of the backlog, they should get additional detail. Ideally, at the point when they are pulled into a sprint, they should already have sufficient detail to estimate the effort required to finish the task, and as much detail as is necessary so that everyone on the team has a good understanding of what the solution will look like in the end. Certainly, the solution will be refined as it is developed (which should be documented in the story’s description or comments). Various team members may be asked to flesh out the details on upcoming stories, so that they are ready to go when they are pulled into a future sprint.

Sprints

Sprints are a repeated chunk of time -- 2 weeks in our case -- that form the cadence of our ongoing development. We move issues from the backlog into the sprint to be worked on. A sprint starts with the Sprint Planning Meeting, has Daily Standups as we go, and ends with a Retrospective. More details on those below.

Sprint Planning

At the Sprint Planning meeting, our goal is to determine which issues will be worked on in the upcoming Sprint. The team works from top-down in the Release Backlog, pulling issues into sprint, fleshing out details as necessary, and assigning team members. Issues are pulled in until everyone is expected to be comfortably occupied for the course of the sprint. Ideally, issues are pulled in strictly from the top of the backlog, but accommodations may be made to balance team member’s individual workloads.

The preceding sprint is first closed out before planning the upcoming sprint. This involves closing out as many of the open issues as possible, and moving the remaining open issues to the upcoming sprint. The stories should also be reviewed to understand why the work was not finished, and if any action needs to be taken to make sure they don’t just roll through another sprint. (It is normal that some work simply remains to be done because there just wasn’t enough time to accomplish it.) It is assumed that these issues are of higher priority than any new work (otherwise they wouldn’t have gotten into the preceding sprint in the first place).

Workflow

To Do

Issues begin in the “To Do” state. When a team member commences work on the issue, they should move it to “In Progress”. For odd tasks that don’t really have a “development” stage, they should move straight to the “Testing in Progress” stage (for example, a task to verify that the software works with newly-released OS versions or hosts/plugins).

In Progress

While a developer is fixing or implementing an issue, it should be in the “In Progress” state. When a developer completes the issue, they move it to the “Ready for Testing” phase. Assign it to the appropriate tester, if known. (Assigning it to a tester will speed the process, although testers will also sweep for issues in the “Ready for Testing” phase that are not assigned to a tester.)

The developer must ensure that there is a means for the tester to verify the issue. Usually, this is simply via the user interface -- in which case, no special handling is needed. If that is not possible (e.g., it involves some new back-end work that is not available via the normal user interface), the developer must figure out another way to verify: make a canned test available via the Debug menu, or unit tests, or some other test using an automation framework, or even just another developer stepping through in the debugger.

The developer should certainly test their own work before handing it off to the tester. The developer should not rely on the tester to smoke out obvious issues that can be found by simply running the obvious scenarios. Certainly the developer can rely on the tester having access to a broader array of OS versions and test data, but the developer should have a reasonable expectation that they are producing quality software prior to the testing stage.

For odd tasks that don’t really have a testing stage, the issue can skip to “Done”. This should be used sparingly, and only if there truly is no point in having another person double-check that the desired outcome has been achieved.

Ready for Testing

Issues that are dev-complete accumulate in Ready for Testing. When a tester begins working the issue, they should move it to “Testing in Progress”.

Testers should regularly look for issues in the “Ready for Testing” state that are not assigned to a tester, and assign them as appropriate.

Testing in Progress

In this state, a tester is actively verifying that the purported solution fulfills the criteria specified in the issue, and that it does not cause unintended side-effects elsewhere.

If some fundamental problem prevents the tester from even starting to test the issue, assign it back to the developer and move it back to “In Progress”, with a comment on what is preventing testing. (Optionally send a sternly-worded, pointed email to the developer asking if they even tried running it at all.)

If a problem is discovered during testing, do one of the following: for simple issues (i.e., the issue is only a single thing that manifestly either works or it doesn’t), assign it back to the developer and move it back to “In Progress” (with a comment describing the problem). For more complex issues: open a new bug, schedule it in the current sprint, assign it to the developer, and link it as “blocking” this issue. This is especially important if multiple problems are found, so that they can be tracked properly.

An issue cannot move past the Testing in Progress state until the implementation satisfies all the requirements of the issue, all of its “blocking” issues are themselves moved to “Done”, and there are no unintended side-effects. (To be sure, the tester cannot regression-test the entire program on every issue, but a certain level of care should be taken to test around the issue in areas that are likely to be affected.)

A story should not be moved to “Done” until the code is in a shippable state. There are two aspects to this: first, it should meet our standard of quality. I.e., whatever it is purported to do, it does correctly and without unintended side-effects. The other aspect is, as a matter of completeness, would we want to release it to the market in its current state? The standard of quality must always be met before moving a story to Done, but the standard of completeness has a bit more flexibility -- if this story is just one component of a larger series of work, it does not necessarily have to represent a “complete” solution, i.e., one that we are ready to send to the market. Some consideration should be given to “hiding” the incomplete implementation behind a Debug option (or similar), although this is not strictly required. Certainly it should be whole as far as it goes, i.e., no dangling buttons that aren’t hooked up to anything, or other such things.

If much worthwhile implementation has been accomplished, but some final wrinkle is preventing the issue from moving to “Done”, we have the option of splitting the story. To do this, open a new story to cover the hard case, so that the rest of the work can be verified and closed. This should only be done if the completed portions of the story form a valid, whole, coherent story on its own. We should not split a story as a way of ducking a problem or kicking it down the road.

If there truly is a show-stopping problem, we should either resolve it, or find a way to set aside the incomplete work (i.e., disable/hide it) until the problem can be resolved.

Done

An issue should not move to “Done” until it is truly done -- implemented and verified, with no unintended side-effects. “Done” issues are accumulated into Release Notes for the release it is assigned to. At this point, the issue enters into the historical record and should no longer be edited.

Special Situations

Problems Found After “Done”

Of course, it is possible that problems will be found with a story after it’s been moved to “Done”, perhaps because something was overlooked, or because someone remembers some other scenario that really ought to have been part of the story. If we’re still in the same sprint as when the work was completed, go ahead and undo the change to “Done” and carry on as if it never was marked as “Done”. But if the story is “Done” and the corresponding sprint is past, leave that story as-is. Instead, open a new story with a description of the new requirements, and link it to the previous story as “relates to”.

Regressions

If something that had previously worked is now broken, write up a bug per usual. Regressions should go to the top of the backlog and assigned to the current release. Regressions should be addressed before starting any other new work -- ideally by fixing them, but otherwise by coming up with some other plan to mitigate them.

If it can be determined that some in-progress work caused the regression, link it to the new bug as “is blocked by”. Don’t do this for work that is in a previous sprint and already marked Done -- instead, just link it as “relates to”.

Closing Issues for Other Reasons

If an issue is written up and it is discovered that it had already been written up, this new issue should be closed out as a duplicate. Link it to the other issue as “duplicates”. Move it to Done, and clear out the Fix Version field (the Fix Version field must be cleared to prevent it from showing up in the release notes). If there is useful information in the duplicate issue that was not present in the original write-up, incorporate that info into the original issue, either in the Description or Comments.

If we decide that we will not take action on an issue, perhaps because we could not reproduce it or we decided to go another direction, do the following: add a Comment that describes the decision, clear out the Fix Version field if necessary, and move it to “Done”.

Note that we do not use the “Resolution” field. If something is Done and has a Fix Version, that means the issue was implemented. If it is Done but does not have a Fix Version, that means it was closed without taking action (and see the Comments for why).

Daily Standup

On a daily basis, each team member in turn answers these three questions:

- What did you do since last time?
- What will you do until next time?
- Is there anything in your way?

As each team member answers these questions, the other team members have an opportunity to see how the work affects them -- perhaps it will come to them next in the workflow, or it may conflict with something they are doing. It provides an opportunity for anyone on the team to say “wait, what?” in case something is going off the rails or hasn’t been communicated effectively to the rest of the team. If a team member reports that something is in the way, the team should figure out what can be done to remove the block (this may involve an additional meeting or conversation outside of the Daily Standup).

The Daily Standup should take no more than 15 minutes. If there are bigger issues to discuss, they should be deferred to another meeting or conversation with the appropriate personnel.

Our Daily Standup takes the form of a shared Google doc and a calendar reminder to have it updated by 11:00 Eastern:

https://docs.google.com/document/d/1FsjPKUhS__7cxoF1mD0rmsUWoCcJ-xox7u5BvSrVi20/edit

Retrospective

At the end of each sprint, we meet to have a Retrospective. At the Retrospective, the team collaboratively answers these questions:

- What went well?
- What went poorly?
- What should we do differently?

The goal of the Retrospective is to do continuous process improvement around our development methodology. Any aspect of our process is fair game for improvement, and we should have a bias towards experimentation. The meeting should produce particular Action Items assigned to individuals to address issues that arose during the discussion. Our Retrospective takes the form of a RingCentral meeting on the last day of the sprint, with the outcomes recorded in a Google doc:

https://docs.google.com/a/alienskin.com/document/d/1Voh8SQuisEYPIbqY-xzLuzFMetpUrJCIYIJOVyD1jMI/edit?usp=sharing

Scrum Master

The "Scrum Master" role is held by a member of the team. It is more of a bookkeeping role than a supervisory role (ideally, the team and the individuals on the team should be largely self-managing). Duties include:

- Facilitates the usual meetings (planning, retrospective, daily standup)
- Runs down blocker issues
- Works with broader organization (management, marketing, etc.) to define the backlog
- Keeps backlog "groomed" (refining details & estimates with the help of others as necessary)
- Triages new stories and bugs as they come in
- Prepares reports for the broader organization
- Serves as a resource for how to solve process problems in an agile fashion
- Drives process evolution