

Transforming a Six Month Release Cycle to Continuous Flow

Matthias Marschall
conject AG
matthias.marschall@conject.com

Abstract

This paper tells the story of how the team managed to speed up delivery whilst raising quality at the same time. conject used to release complete modules of their Internet platform once every six months. Customers were forced to wait for months to get access to new features. Once they arrived, the big bang releases disrupted the user experience due to bugs lurking in the new software. Despite what seemed to be an impossible task, the team successfully transformed their software development process to deliver a stable and continuous flow of small releases. Now features are delivered to customers more quickly and with much higher quality.

1. Introduction

1.1 conject.com

conject is building and operating an on-demand business services platform for Infrastructure Life-cycle Management (ILM) in the Real Estate Industry. Hosted on the Internet, the modules of our platform deliver industry specific solutions, enabling traceable collaboration and providing a centralized document repository.

1.2 The Entrenched Team

Many of the challenges we face with our platform come from the history of how our platform was developed in first place: The first two major versions were implemented by an external team and whose initial idea was to build a collaboration platform SDK instead of an industry specific on-demand platform. By the time my team took over from the initial creators it was confronted with major difficulties. The business demands were radically shifting towards requiring industry specific functionality while the team struggled to get their arms around the existing legacy codebase - a 300 KLOC beast.

2. The Six-Month Release Cycle

2.1 Pizza Releases

At first, our release strategy involved implementing one or two complete modules in a six-month time box and then spending a whole night doing a big release including any last minute bug fixes. These long release nights were nicknamed "Pizza releases" due to the fact that we ordered pizza for dinner, which was paid by the company.

Before each pizza release we had a two to three week testing cycle, during which the customers tested the platform and the development team fixed bugs and "finalized development". Development finalization included data migration scripts and those "last" change requests that "must be in this release". During these test and fix cycles the number of open issues always increased faster than we were able to resolve them. Our issues backlog continued to grow with each release.

This release process led to problems with quality and morale. The quality of the modules suffered, as we had to stop fixing bugs in order to release in time! The morale of the team decreased due to the high pressure of delivering "yet another feature" on time and the never-ending growth in issues.

Nevertheless, these big, infrequent releases with their multiple rounds of regression testing (none of which ever was really complete due to hot fixes and change requests) gave us the warm and fuzzy feeling of having done everything possible to ship the best product. Given the feature requests we were receiving and the resources we had, we were doing the best we could and we really thought that there was no way to get more quality out of our product.

2.2 Emergency Releases

A paradox, which we tried to ignore, was the fact that on the one hand we considered big testing cycles fundamental for releasing anything but on the other

hand we did releases for emergency bug fixes multiple times a week. These releases did not have any regression testing, only the fix itself was tested briefly. And it didn't even matter how complicated the fix was - we released it when we thought we had to. Of course, doing this caused undesired side effects, which often required yet another emergency release.

The platform and my team were in really bad shape. The quality was not as good as it should have been, we needed months to release features that our customers needed yesterday and our code base was very hard to change. We needed to change something...

3. Running Tested Features

Looking at our ever growing issues backlog and the way we ran our releases, what bothered me most was the pile of unfinished work we were dragging into the test and fix cycles. The developers were trying to finish tasks when they should only have been fixing critical bugs. All testing by customers was *only* done during test and fix cycles. This customer testing consisted of a manual process that took a long time and was quite unreliable. The last minute code changes made life hard for the customers, as they didn't have enough time to thoroughly re-test all necessary areas.

I felt that if we really finished our user stories *before* starting the testing cycles then there would be more time to find and fix bugs, and so we would improve quality.

Looking for a way to approach this issue I came across Ron Jeffries' article about "A Metric Leading To Agility" describing the idea of "Running Tested Features" (RTF), which helps teams focus on delivering features [1].

3.1 Getting Started

Before introducing RTF, developers committed their code to the main line of development at any time. Testing was done in the test and fix cycles every six months. When we introduced RTF, our first change was to require a customer sign-off for each user story before it could be considered *Done*. At this stage we still continued to commit to the trunk whether or not stories were *Done*.

To better visualize our adoption of RTF we introduced a story wall tracking the status of each story in development with the states *New*, *In Progress*, *Dev Complete* and *Done*.

3.2 The Bottleneck

Interestingly enough most stories clustered in the *Dev Complete* state on the story wall - we obviously had a bottleneck in getting stories signed off. Developers just finished coding, moved the story card to *Dev Complete* and then went on with the next story at hand - they didn't really care whether anyone was there to do the sign off or not. Investigating this situation I came across a very interesting opinion most of the developers shared.

3.3 Testing Is None Of My Business

Developers were focusing on making their code run. As soon as it appeared to work they declared that task *Dev Complete* and (from their perspective) thought they were done. Everyone saw in the daily stand-up meetings that hardly any story was moved to *Done* due to lack of sign-off testing, but the developers did not feel any responsibility for this. It was the customers' issue to deal with the sign-off problem, right?

One day I became so sick about that attitude that during a stand-up meeting I walked over to the story wall and tore off the *Dev Complete* stage! I explained to the team that *Dev Complete* is of no value to our customers and is nowhere close to being *Done*. I tried to convince them that it was *their* responsibility to drive a story to *Done*. They had to make sure that it was signed off in the end. *Dev Complete*, I stated, is nothing other than *In Progress*.

That measure finally did the trick for us - developers realized that they wouldn't earn any story points as long as they were not taking care of getting their stories signed-off. From that day on, the developers took great lengths to get hold of customers to sign-off their stories. We were one step closer to really getting things *Done*!

3.4 I Can Do This Later During Testing

Despite introducing RTF, our pile of open issues continued to grow with every release. Looking closely at what the developers did during the test cycles, I observed that they were not just fixing unintended side effects and bugs, but that they were also working on data migration scripts, translations and other integration work that they had failed to address during the user story implementation. These tasks weren't considered important enough to be required for a sign-off and the customer doing the sign-off didn't really care about such "technical details". As all the sign-off testing was done manually there was a lot of room for interpretation about being *Done*. Stories were being signed-off even though important tasks were

outstanding – these tasks were then spilling over into the test and fix cycles.

3.5 What Is Done?

In a release retrospective I mentioned these observations. This started a heated discussion about what *Done* meant to us. There were different opinions ranging from “*Done* means the code has been committed” to “*Done* means the code has been released into production”. After some time the majority of the team agreed that *Released* was the real *Done* of any story. But there was significant reluctance to using this definition in practice as our multiple-month release cycles would mean that we would not be able to declare a story as *Done* for several months!

4. Continuous Releases

The team continued to struggle with the dilemma of not having a commonly accepted definition of *Done*. It seemed that we reached a dead end in improving our process.

It was at this time that we started talking about immediately releasing a story as soon as it was signed-off. First, we only joked about it. As the discussions became more serious over time, we started to see that we were basically already doing just that - releasing stories as soon as were ready - with our emergency releases. Why not apply the same practice to all stories?

It took some more time and discussions to analyze the impact and risks of dropping our regression test cycles and just relying on well-tested stories. On the one hand we didn't see too much value in regression testing as it was always interrupted by code changes and the end result was not so much a stable product as a growing backlog of issues. On the other hand we recognized that our sign-off tests for individual stories might miss negative side effects in areas not expected to change at all. We knew this might cause us problems when switching to continuous releases.

4.1 Dropping Pizza Releases

After pondering continuous releases for a while the release team finally approached me and proposed to try it out. They said that it wouldn't be any worse than the current strategy and that it had a good chance of improving our quality in the long run. We cancelled our upcoming “Pizza-Releases” and started adding regular stories to the former “Emergency Releases” instead. Doing that meant that we dropped our test and fix cycles. As our new releases only contained a couple

of user stories, the risk and impact was much lower, so we stopped asking the whole team to stay and the release team did their work from home.

Our release strategy had now been transformed: we were continuously releasing every two or three days, not just in emergencies, but whenever we had some stories to release.

I never get any free pizza anymore now that we're doing these continuous releases – Developer

4.2 Developer Responsibility

When the release team and I confronted the developers with our new process - releasing a story as soon as it is signed off - it scared the hell out of them. Suddenly they realized that there was no room for mistakes anymore. The air got a lot thinner. A serious bug by a developer could cause the whole release to be rolled back!

Finally we had found a way to communicate to all team members what we expected from a story that was marked *Done*: No further work should be necessary, it should be releasable. This made developers think twice before declaring any story *Done*.

Developers started to take much more care for analyzing possible negative side effects during coding. Additionally the Customers put more focus on providing us with test scenarios, which again ensured the absence of unintended side effects. Everyone was aware that not doing impact analysis was a great risk as there was no security net of a full regression test before a release any more.

4.3 Grace Periods And Broken Flow

We continued to release these mixes of urgent bug fixes and regular stories every couple of days for several weeks and it worked out pretty well. There were quality issues from time to time but overall we were pleased. However, we had some unresolved large pending releases – we had been working on some technical refactorings for a few months, and these were now ready to release. The release team was expecting some trouble after such significant releases and so they decided to introduce a grace period of one to two weeks to let things calm down before new stories could get pushed out. The release team was worried that if they allowed too many concurrent releases, then they would find it difficult to identify which release was causing which problem (and the potential risk of side effects would be even greater).

This seemed like the only way of dealing with our backlog of huge changes, but it caused some additional

trouble. In fact, these big batches left over from the big-bang release days were serious enough to require full regression testing and major releases. But we did not want to fall back to our old pattern of releasing. We wanted to force ourselves to keep to small releases and we believed it important enough to fight for.

Eventually, we put a lot of effort into sign-off testing for these big batches and we released them without test and fix cycles, as if they were a big story. Having the grace period after the releases helped to stabilize the platform very quickly.

4.4 The Quality Debacle

What we did not anticipate was the extreme negative effects of building an inventory of signed-off but not released user stories during the grace periods. I expected that even if we release a bigger batch of Running Tested Features there should not be any problem. But I was wrong.

Releasing 12 to 15 independent stories at once created a stability nightmare. The day after the release we experienced five times more errors than average. Additionally customer support registered three times more inbound calls after such releases.

These quality problems put the whole team into fire fighting mode - fixing bugs as fast as possible and then releasing the bug fixes immediately. Under such pressure the team fell back to it's old ways of not getting such fixes signed-off and not analyzing possible impacts thoroughly enough. The outcome was devastating! Every release fixed a couple of bugs but introduced a bunch of new ones that needed immediate attention. We were in a vicious cycle. After a couple of weeks even our own customer support dreaded releases.

The situation was out of control. The quality of our product was worse than ever. There were voices demanding to fall back to our old release strategy. Even within the development team we thought that it might help to do releases less frequently, perhaps once every two to three weeks instead of multiple times a week. But I had the strong feeling that lengthening the release cycles again wouldn't improve matters. I dreaded losing the benefits we had seen from continuous releases – the flexibility and the feeling of responsibility that the developers had shown.

4.5 STOP! Slow Down!

In trying to understand why our releases had failed so miserably we discovered that we were developing stories faster than we were able to release them. The release team was swamped with fire fighting and trying

to release new stories at the same time. Under pressure, the release team was not verifying the quality of stories given to them for release. And as all sign-offs were still done manually the quality varied widely.

To break the vicious cycle we took a drastic measure: we decided to only release stories that had accompanying automated unit and acceptance tests. This was a major change for us since we had at this point only done some prototyping of test automation - it was not yet a part of our process. Suddenly every developer had to jump-start writing unit and automated acceptance tests.

Introducing automated testing stopped the flow of releases for a couple of weeks. But our buggy releases had conditioned the users to be fearful of releases: they didn't complain about not getting new features for a while. They were glad, at least for some time, to have a stable platform with all its *known* bugs and shortcomings.

4.6 The Quality Gate

Introducing automated testing slowed down the flow of new stories but the stories were of a verifiable quality. We slowed down because we 1) had to learn how to do test automation and 2) build testing infrastructure. The testing ramp-up overhead had a significant impact: a fix, which might have taken half a day before automated testing, could now take three to five days due to the testing ramp-up overhead. Despite the significant cost increase, we were ready to pay that price, since we felt it was the only way to deliver quality features.

To enforce sufficient test coverage for every story we introduced a quality gate. We ensured that our goals for required test coverage were met by creating automated reports from the unit test coverage tool EMMA [2]. Additionally we required that all changed code must be free of any serious bugs found by the automated FindBugs tool [3]. We made one team member responsible for merging *Done* user stories to the release trunk. This person is responsible for verifying these quality metrics and only merges a story onto the trunk if all of them are met.

Automated testing, code coverage and FindBugs as part of the build cycle was the best decision we've made in years. We know immediately if a new feature or a fix is ready to be deployed to production or needs to go back to the drawing board. - Member of release team

4.7 Developer Smoke Testing

All these measures helped but as the code coverage was almost nonexistent in the beginning and was only slowly getting better the team saw the need to do regression testing to cover all the shortcomings of automated tests. But the team no longer wanted to rely on a team of customers testing - they knew that they would have to do it themselves if they wanted to be sure.

The team asked the customers to come up with a list of the most critical features in the platform. This list is now the basis for a manual smoke test executed by the development and release team before any complex release. The team finally took full responsibility for quality without relying on anyone else.

5. Smooth Continuous Flow

Our experience with big batches of user stories showed us clearly how crucial a stable flow of small releases is. As we released sooner, and without the test and fix cycles, the quality problem hit us much harder and earlier than before. But it forced us to really address the root cause of any issue to avoid it happening again in the future. Having buffers hid a lot of problems and did not force us to resolve others.

Every disruption in the flow causes batches of unreleased user stories. And trying to release such batches resulted in over proportional numbers of errors after the release. We really became very conscious about keeping a continuous flow to avoid all these problems. Only in keeping releases small by avoiding big batches did we see a chance to keep release quality high.

Today, we go to great lengths to release big refactorings or sets of dependent user stories step by step even if such releases don't make sense from the user's point of view. They make sense for us because they preserve the flow and avoid big releases.

One example of successfully breaking a big refactoring into a series of small releases is our transition of our platform to support Unicode characters. Instead of doing all code changes and reinstalling the database at the same time, we added Unicode support to one area at a time only and released each area on its own. Making the database aware of Unicode characters was the very last step in the process. Finally, our international customers could use all the previous, preparatory releases.

I begin to get an uncomfortable feeling if we haven't released in over a week... - Member of release team, who demanded to shift to three-week release cycles earlier

5.1 SVN – One Branch For Every User Story

To avoid blocking the release flow we have decided to keep the trunk – our main code line for releases – in a releasable state at all times. We know that if we let non-release quality code into the trunk then the next release will be blocked (until fixed). Such blockages would lead to a backlog of developed but un-released stories.

We also want to avoid large merges into the trunk, as we have previously seen that large merges cause stability issues. So we have decided to create a separate branch for each user story. A story only gets merged into the trunk once it has successfully passed the quality gate. After merging into the trunk the story normally gets released on the same evening.

We have found that this practice requires every developer to regularly merge from the trunk into their own story branches to avoid their branches getting too far away from what's currently running in production. To help avoid this, we have introduced a small tool that sends a reminder email if a branch has not been updated recently (within the last week). This helps us to avoid “zombie” branches, i.e. branches that are completely outdated that would be and difficult to merge into trunk.

5.2 Test First - Executable Specifications

Another cause for unwanted variations in the flow is re-work, i.e. features that need to be changed after their release, either due to bugs (broken functionality) or insufficient understanding of user requirements (wrong functionality).

Until recently, the customers used to communicate user requirements to the developers by creating a presentation describing the rough outline of a solution for a user story. At best, such a presentation contained some screenshot fakes. There was hardly any face-to-face communication or discussion of the concept between the customers and the developers.

Usually the developers started coding based on these slides while the customers started to note down some rough test scenarios in a spreadsheet. This spreadsheet they kept for themselves until they started sign off testing. During the sign-off testing they constantly delivered bugs and missing implementation details to the developers.

To boost communication and to avoid misinterpretation and ambiguities, I asked my team to write automated acceptance tests using a FitNesse wiki [4]. To make these tests helpful for defining what the user story should do required us to come up with a writing style that would be easily readable and writable

by our customers as well as developers. We achieved that by using DoFixtures, which allowed us to create FitNesse tables that read like natural language [5].

The tests act now as specification by example and they build the basis for a deep discussion between the developer and the customer about the details of a user story. Such dialog used not to happen too often without both of them being forced to work together on automated acceptance tests.

Today the customers and the developers create a shared understanding of all the details of a user story by working on real world examples codified in the automated acceptance tests. This not only ensures that no features break on future changes but prevents the blocking of the release flow by having to re-work already released stories to a large extent.

5.3 Continuous Integration

The last building block for ensuring flow was the introduction of continuous integration into our release process. After deciding that our trunk should be releasable at any time, we became aware of the need to *know* whether it was in that state or not. Instead of running our test suites manually from time to time we decided to setup CruiseControl [6] for automatically building and testing our application whenever a commit to the trunk happened. Having CruiseControl watch the trunk and notify us whenever something was broken made us aware of any trouble immediately after it occurred instead of letting the trouble show up when we had to release something. This avoided fire-fighting activity and helped to smooth the flow.

Although it was good that we could see problems on the trunk sooner, we wanted to avoid these problems altogether. The development on the branches was still being done in a “mini waterfall” model. This involved separate phases of developing code, adding tests, running test manually and collecting test results, and adding test results to an email for requesting a merge to the trunk. This process of manual work combined with doing the tests after the coding produced peaks of workload for the teammates managing the trunk. They had to inspect all the manually gathered test results and decide whether or not the quality was sufficient to merge the code to the trunk. Furthermore the developers were burdened with running all the tests manually and collecting the results.

To avoid the manual work as well as the peaks we decided to extend the usage of CruiseControl to all development branches. To make this process efficient we made a small change to CruiseControl to enable coverage of any new branch right from the point in

time when it was created. By running CruiseControl on every user story branch created we have found that the developers are motivated to keep their branch clean and tested at all times (as no-one wants to see a build failure email). Developers were now really adopting test first approach for story development. Additionally the need for running tests and collecting results manually was gone as it was always obvious whether a branch was in release quality or not. This measure extended the smooth flow from the trunk to all branches being worked on simultaneously!

6. Summary

We have seen that deferring testing and releases for months and mixing the work results of several developers causes the developers to feel disconnected from the results of their coding and leads to developers feeling little or no responsibility for what they create. When developers are not able to make a direct connection between their contribution and the results they tend to loose interest and get frustrated. This has led to low product quality in our case. We have seen that developers show more commitment if their stories are released as soon as they declare them *Done*.

We have learnt to focus on keeping a smooth continuous flow of small releases, to avoid the problems caused by releasing too many stories at a time. To maintain the smooth continuous flow, we have learnt to take great lengths to break bigger features or refactorings in multiple smaller pieces, which can be released independently.

Acknowledgements

Thanks to Mike Hill and Dan Ackerson for help writing and editing this paper.

References

- [1] R. Jeffries, “A Metric Leading to Agility”, <http://www.xprogramming.com/xpmsg/jatRtsMetric.htm>
- [2] EMMA website – <http://emma.sourceforge.net/>
- [3] FindBugs website – <http://findbugs.sourceforge.net/>
- [4] FitNesse website – <http://fitnesse.org>
- [5] J. D. Miller, “Create a Testing DSL with FitNesse and Selenium”, <http://codebetter.com/blogs/jeremy.miller/archive/2006/07/15/147400.aspx>
- [6] CruiseControl website – <http://cruisecontrol.sf.net/>