2013 46th Hawaii International Conference on System Sciences

Release Duration and Enterprise Agility

Daniel R Greening
Evolve Beyond
dan@greening.org

Abstract

Short release duration—the time from starting development until it delivers measurable value (i.e., paying customers adopt an upgrade)—is an implied goal of agile methods. Release duration incorporates the expensive parts of the value chain: build, test, deploy and sell (but not exploratory design, for example). Release duration correlates with technical debt. Attempting to reduce release duration may help drive agile behavior through a company. Finance departments often collect release duration, helping a company assess its agility.

Citrix Online illustrates how process methodology, development group size and release duration relate. Its adoption of Scrum and Enterprise Scrum drove release duration down from a peak of 41 months to less than 4, shorter than it had as a small startup. Its market share rose during the same period. Data from another company, PatientKeeper, also seems to indicate that short release durations correlate with more profitable outcomes.

1. Introduction

Agile processes like Scrum help development teams adapt to markets, gain engineering efficiency, and forecast releases more accurately than traditional waterfall processes. Software engineering is an expensive and risky creative activity. While software projects can generate dramatic returns when successful, they often experience unanticipated delays and have a high failure rate [char2005].

Short product development iterations (called 'sprints') are a hallmark of agile teams. Scrum teams produce a releasable product *at least* once a month. Agile coaches typically encourage teams to set definitions of what "done" means for completed features that increasingly approach the holy grail of "delivering to the customer."

Frequent deliveries, assuming that users can keep up with new features and provide feedback, enable product managers to better test market theories to maximize profit. Steve Blank and Eric Ries [blan2005][ries2011] pioneered an approach I call "lean product management." They assert product managers can more accurately forecast value and maximize profit by developing and delivering test features that validate value assumptions, striving to cheaply determine whether customers will pay for a feature, respond to a marketing channel, fulfill through a particular distribution channel, etc. When these experiments validate an approach, product managers can invest further in software development. When experiments invalidate a market, product managers can "pivot" development efforts to address more promising opportunities.

Too frequent delivery might be disruptive for lean product management. If release cycles are too short, early-adopters can't use the release and provide feedback rapidly enough to affect later releases, significant changes can disrupt user workflow, and a release's short operating time can mask errors that take time to appear [cope2012]. These problems can be resolved with some creativity in performing a release. For example, in 2007 PatientKeeper was able to deliver 45 releases to users. In most of those releases, changes were incremental and non-disruptive to PatientKeeper's physician users. When a release introduced major changes, PatientKeeper would first deliver the release to a subset of users, get feedback and then deploy to the rest [suth2012].

Frequent deliveries compel engineers and designers to better identify and mitigate deployment and usability problems. Traditional waterfall approaches proceed from a design phase, to a prototype phase, a development phase, a testing phase, a deployment phase, and finally to a maintenance phase. Design decisions or the nature of the product itself can cause deployment or usability problems that won't appear until the product is deployed or used. In some cases, these problems could doom the product. Most companies would like to know early if a product is doomed, so they could spend their money on developing a more profitable product. But long release durations can mask these problems until after the company has squandered funds on development.

Frequent deliveries can motivate engineers to implement automated testing. Squeezing the release

1530-1605/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2013.463

IEEE computer society

process into short durations can force teams to automate testing. Automated testing usually reduces the long-term cost of testing: the cost of automated testing amortized over multiple releases can be much cheaper than manual testing. Automated testing allows programmers to make more significant architectural changes with less fear they will introduce serious bugs. This accelerates development of new features that might require pervasive code changes (examples: internationalization, public APIs, deployment on new devices, identity federation, parallelization, fault-tolerance, etc.). In slowly changing code bases (i.e., for a cash-cow product), automated testing makes it possible to fix bugs for customers at low-cost. Finally, some types of integration tests can also be used as diagnostics for running systems, increasing service reliability.

Frequent releases in online services can motivate architects to implement high-availability architectures. Frequent releases mean frequent deployments, potentially disrupting users more often. The lowest possible user disruption occurs when users are migrated while running an old release to a new release with no disruption at all. To achieve this, one could use an active-active rolling upgrade approach. Motivated to keep release duration low, I designed an active-active rolling upgrade architecture for Citrix Online's first Scrum project. Citrix Online later used the same approach for other products.

2. Technical Debt

"Technical debt" is perhaps the major factor that causes release duration to increase. Ward Cunningham coined the term to describe code that remains when programmers sacrifice long-term productivity for (perceived) short-term completion speed. Here are some common examples:

1. Sometimes a programmer copies existing code, pastes it somewhere else and modifies it to satisfy a new requirement. In the short term, the programmer has fewer worries about introducing bugs into old code, and avoids having to write new code.

   Copy-paste technical debt can increase future release duration. An existing bug could have been copied from the original code, creating a new bug in a different place. When a user encounters one of the bugs, a programmer might only repair only one. The cost of repairing a bug after release is much higher than avoiding bug creation or repairing the bug before release [jone2009].

   A copy-paste approach usually increases the code size. If developers write unit tests, to maintain the same level of code coverage, they must write additional unit tests, build times will increase and test maintenance costs will increase.

   Instead, the programmer could have refactored the original code to handle both requirements, possibly using a shared method. If a bug was retained from the original code, it will likely continue to exist in a single place. The size of the code is not likely to increase as much as it would with copy-paste.

2. Sometimes different teams "fork" a code base in a source code repository, essentially making two copies of the code. They then make changes independently, intending to merge them later into a single copy. Programmers can then worry less about conflicts in code changes, and avoid delaying an impending release.

   Code-fork technical debt can increase future release duration. The theory that code changes can be easily merged is often proven false. I've actually never seen this approach work well. In a different company, a team forked a code base and then abandoned one of them when the changes proved infeasible to merge, at a labor cost of about $1 million.

   The tendency to fork code often arises when automated testing is not sufficient to assure programmers that changes made by others won't disrupt their work.

3. Sometimes teams develop code without corresponding automated behavior (black-box) tests. Then, to ensure a high-quality release, the code must be manually tested. Automated behavior tests take time to develop, so programmers seeking to release earlier or with more features often believe that manual testing will be faster (or just don't like writing automated tests).

   Manual test technical debt can increase future release duration. If programmers continue to develop code they worked on in a previous release, they can easily create bugs in functionality that previously worked (called "regression bugs"). Thus, on every subsequent release many of the same manual tests must be repeated, to ensure a quality release. Over time, as functionality increases, the manual testing time can easily eclipse the time for developing new features, making release duration unreasonably long.

For example, if a team develops code for a new mobile device, it may reuse much of the existing code (hopefully by refactoring, see above), changing only what is necessary to support the new device. However, even if very little has changed, functionality on the old device must be retested.

4. Sometimes customers ask to continue to use older versions of products. Installing new releases can disrupt users and can introduce new bugs. When this occurs, companies must support different versions.

   Multi-version support is a form of technical debt. When bugs are found in the most current release, older releases may need to be checked and fixed. With each additional version supported, more work is required to fix any bug. With each bug-fix release for each version, regression testing may be required.

   Developers have a few alternatives, depending on their customers. They can use a software-as-a-service model, so customers always use the latest version. They can force customers to upgrade before taking a support call. But in some cases, the customer requires long-term support for old versions; this will increase release duration over time.

These examples illustrate the technical debt concept. Programmers purchase short-term speed-ups or customer advantage by increasing future release durations. This approach parallels how some people get into intractable credit card debt: they buy things that improve their lives for the short term, while mortgaging their future. Sometimes buying speed on credit makes sense for developers, such as very early in a startup company's lifetime, when the market hasn't yet been proven; with proven markets where competitors loom, technical debt can be much more problematic.

Technical debt is one of the most prominent reasons many companies have difficulty reducing release duration while retaining the same quality.

3. Finance tracks release duration data

Teams themselves may not track how often their work reaches a customer, but the finance department likely does. Software development is a form of asset creation. A company usually invests the most development in a software project early in its lifecycle. As long as operating environments don't change, the same software could earn revenues or cost-savings over many years with few additional expenses.

Finance departments typically track the dates important to release duration. When a company starts investing in software development, and before that investment can start producing value, it starts "capitalizing" the software development as an unused asset. Once software goes into production and earning money, a company then starts "depreciating" the investment over the productive life of the software, as an expense. The difference between these two start dates, is the release duration.

Few agile companies release software to customers after every team sprint. Larger companies often have multiple teams working on a public product release: combined testing, configuration and deployment for the assembled work produced by multiple collaborating teams may take time and additional iterations. However, for all the reasons we discussed, more frequent releases to customers can be a strong indication of a healthier engineering group.

An example illustrates how financial tracking neatly handles software development edge cases: A company develops a software product as a free beta product, delivers it to users and gets feedback. When product development starts, the company starts capitalizing the development cost as an investment. Because it is not yet productive, neither earning nor saving money, depreciation does not yet start. Only when the product becomes productive would the company start depreciating the asset.

This fits perfectly with the lean product management approach, which encourages companies to make users pay even for beta products. Requiring payment in a beta release helps product managers obtain more credible profitability forecasts for the final product. Payment software issues can insidiously damage profitability for companies, and our definition of release duration can expose this problem to the light of day.

4. Citrix Online

The development history of Citrix Online demonstrates release duration as an agility metric. Citrix Online began as a startup called ExpertCity in 2001. ExpertCity used waterfall methods to develop screen-sharing and conferencing software. It offered services to small and medium size businesses for a monthly fee. In 2004, the company was acquired by Citrix and became an independent subsidiary. It institutionalized its waterfall approach as a RUP variant.

I joined Citrix Online in 2007, and created its first Scrum team. I developed an agile portfolio management technique, called "Enterprise Scrum", to prioritize the company's projects. We started using Enteprise Scrum in late 2008 [gree2010]. I became the head of Citrix Online's Agile Program Office. Two years later, in early 2011, almost all software engineers in Citrix Online were operating under Scrum and Enterprise Scrum, with 44 Scrum teams.

5. Release duration

Asked to demonstrate the benefits of agility to others in Citrix Online, I worked with colleagues to assemble release duration data through the company's history, from startup formation through the end of 2010, and attempt to correlate the data with process changes and organizational events. Throughout its history, Citrix Online has retained its own separate marketing, sales, product management, engineering, finance and IT departments. This independence has made historic analysis easier.

Figure 1. Release duration at different project midpoint dates

Figure 1 shows that ExpertCity's first software project took 10 months from engineering team formation through customer revenue. Its next release duration was shorter, leveraging the functionality it had already built. From that point, release duration gradually increased over time, reaching a pre-acquisition peak of 14 months from project inception to customer revenue.

Citrix acquired ExpertCity between December 2003 and March 2004, and named it Citrix Online. The data point just prior to the "Startup acquired" line is a project midpoint; this project started before the acquisition and was released after it. Much more rapid releases followed this project, likely exploiting additional resources injected into the company by its acquirer. However, following the acquisition we again see a trend that release duration gradually increases over time. During this time, Citrix Online formalized and implemented a RUP-based waterfall project methodology [krol2003]. It did not seem to affect the trend.

I joined the company in October 2007 having some experience with agile techniques in a previous company. One existing Citrix Online project was attempting to merge agile and traditional project methods, and the rest were using waterfall. My team started a new software project shortly after I joined, and we decided to adopt pure Scrum.

Continued missed release targets from waterfall projects, and positive results from agile drove the company to hire Ken Schwaber to train 60 ScrumMasters in March 2008. After that training, projects gradually moved to Scrum.

In October 2008, about half of Citrix Online's engineering teams were following

Scrum principles at least loosely, but upper-level management did not yet completely embrace agile principles. The engineering department faced a long list of projects and pressure to work on all of them. We were spreading engineering talent thin and dragging out release duration. Around this time, few projects were released that gained customer revenue (revenue is a key subtlety, there were non-paid betas released at this time). The average release duration peaked at 41 and 35 months, an alarming state that could enable competitors to gain market share.

Figure 1 shows that, in organizationally stable periods when waterfall methods were used, release duration increased. This seems likely due to the accumulation of technical debt.

6. An explicit focus on release duration

In December 2008, we adopted Enterprise Scrum [gree2010], established 3 months as the desired maximum release duration, measured engineering department velocity, and asked upper management to restrict demands on engineering to the top-priority projects. This began an internally painful period for the company, with much uncertainty and behavioral changes.

It became clear that broad agile training would be required to sustain an agile culture. My team and I provided 2-day agile training in most Citrix Online developer sites. By mid-2011, we had trained 240 employees.

We changed two major aspects of Enterprise Scrum in 2009 and 2010. First, we stopped performing project reviews, retrospectives and planning at a scheduled date every quarter. It was disruptive to engineering staff to plan projects every quarter, particularly when we were not sufficiently agile to be certain that a quarterly end-user release was possible. Instead, we allowed projects to start and terminate any time. This, unfortunately, made it more difficult to thoughtfully track departmental velocity, but reduced context-switching costs.

Second, we realized that many surprise impediments occur within projects, which should have been obvious up-front. We established Project Ready Criteria to ferret out these dangers before projects were approved, and monitor project health while they were proceeding. We developed a method of bulk-estimating a release backlog, which helped teams identify high-risk backlog items.

By the end of 2010, Citrix Online had driven its average release duration to an average of 4 months. This effect was so dramatic that finance staff members raised a concern that their financial projections were rendered invalid: the projections assumed revenue would begin 9 months following project inception, but revenues were coming much faster. When we balanced 'better adaptation to the market' against 'more predictable depreciation', adaptation won.

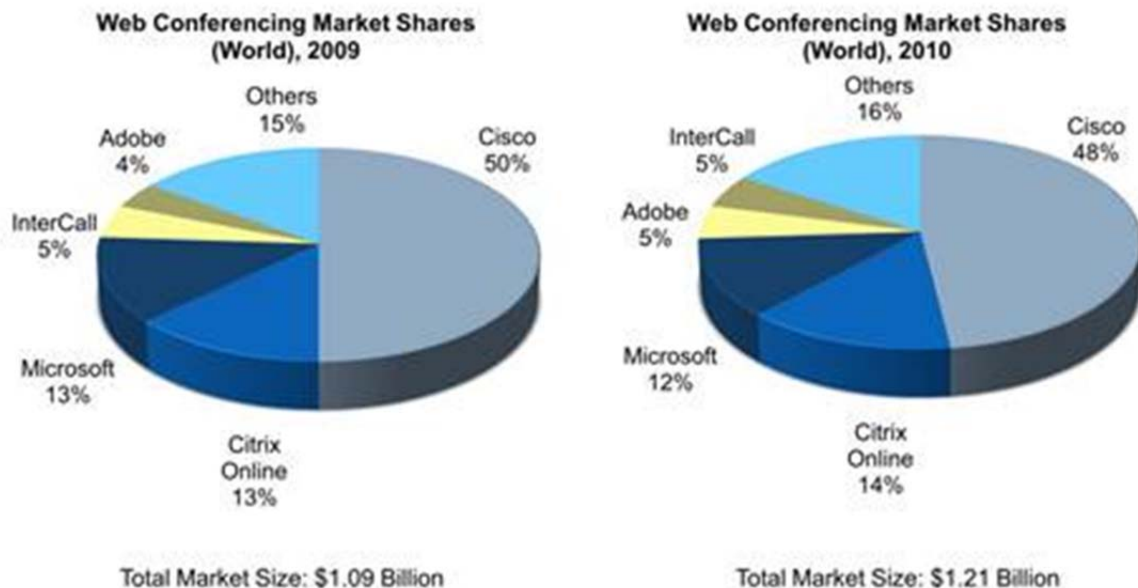

Figure 2. Web Conferencing Market Share, 2009 and 2010

7. Outcomes

In 2008, Citrix Online was #3 in web conferencing market share, at 12%. Figure 2 shows that Citrix Online lapped Microsoft Live Meeting from 2008 through 2010, to become #2, and has begun eroding Cisco Webex's market [fros2011][cox2012]. Citrix Online's adoption of agile methods has dramatically reduced its release duration, and points to a bright future.

Correlation is not causation; this report cannot prove that short release duration helps lead to higher market share. However, similar anecdotal stories appear. Jeff Sutherland (an inventor of Scrum) was CTO of PatientKeeper from 2000 through mid-2008. PatientKeeper's annual revenues rose 400% in 2007, the same year it delivered 45 releases. After Sutherland left the company, PatientKeeper reverted to waterfall and its yearly revenues dropped 50% [suth2012].

8. Conclusion

Managers often claim that startups are naturally agile, but the data given here show startups can rapidly accumulate technical debt, extending their release duration, reducing revenue and increasing cost.

Hiring more engineers can temporarily drive release duration down, at least for small teams. However, without conscious effort, technical debt may then continue to increase, with unfortunate consequences.

A conscious intent to reduce release duration seems to help improve adoption of agile techniques through a company. Three years from its adoption of agile methods and Enterprise Scrum, Citrix Online has driven average release duration from a peak of 41 months to below 4 months, lower than it was when it was an early stage startup. Citrix Online more rapidly adopted agile methods than any other large multi-product company I've encountered.

Release duration is a useful agility metric. It can be easily computed from financial data that many software companies track. It has been correlated with important agility events at Citrix Online. Increasing release duration could point to accumulating technical debt, lurking in a company's code base.

9. References

[blan2008] Steve Blank, Four Steps to the Epiphany, Cafepress.com (Feb 1, 2005). ISBN 978-0976470700.

[busc2011] Frank Buschmann, To Pay or Not to Pay Technical Debt, IEEE Software, November/December 2011 (Vol. 28, No. 6) pp. 29-31.

[char2005] Robert N. Charette,"Why software fails," Spectrum, IEEE , vol.42, no.9, pp. 42- 49, Sept. 2005, doi: 10.1109/MSPEC.2005.1502528, http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1502528&isnumber=32236

[cohn2005] Mike Cohn, Agile Estimating and Planning, ISBN 978-0131479418, Prentice-Hall 2005.

[cope2012] James Coplien, personal communication, 8 Jan 2012.

[cox2012] Dave Cox and Barbara Sanner, Industry Buying Plans & Trends for 2012: Social Learning, Video Training, Mobile Learning and Web Conferencing, Cox eLearning Consultants (January 2012).

[cunn1992] W. Cunningham, "The WyCash Portfolio Management System," *Addendum to Proc. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 92), ACM Press, 1992, pp. 29–30, doi:10.1145/157709.157715.

[fros2011] Frost and Sullivan, Analysis of the Global Web Conferencing Market, 30 Sept 2011, http://www.frost.com/prod/servlet/report-toc.pag?repid=N9D0-01-00-00-00.

[gree2010] Daniel Greening, "Enterprise Scrum: Scaling Scrum to the Enterprise Level," 2010 43rd Hawaii International Conference on System Sciences (HICSS), Hawaii January 5-8, ISBN: 978-0-7695-3869-3 (10 pages), http://www.computer.org/plugins/dl/pdf/proceedings/hicss/2010/3869/00/10-01-01.pdf

[krol2003] Per Kroll and Philippe Kruchten, The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP, Addison-Wesley Professional (2003).

[ries2011] Eric Ries, The Lean Startup, Crown Business (September 13, 2011), ISBN 978-0307887894.

[suth2012] Jeff Sutherland, personal communication, 8 Jan 2012.