Transitioning to Software as a Service: Realigning Software Engineering Practices with the New Business Model

Eric R. Olsen

***Abstract*—Transitioning from software as a good to software as a service is not simply a matter of developing appropriate protocols, adopting new service-oriented technologies, and writing code. Instead, software companies that seek to adapt to a market based on software as a service need to examine how they approach very basic business tasks like marketing and engineering. This paper focuses on the implications a particular software paradigm has on software engineering. It concludes that some traditional software engineering practices in the goods paradigm related to planning, versioning, and maintenance are simply not appropriate in the service paradigm. To be successful as a service-oriented business, a software company needs to realign its base software engineering practices to fit the new business model.**

***Index Terms*—Engineering Culture, Service Engineering, Software as a Service, Software Engineering.**

I. Introduction

Software as a service (SaaS) is an extensively discussed topic in the computing literature today. Most papers are technical in nature, covering topics ranging from solving specific web services problems [1] to more general architectural explorations of the software as a service paradigm [2]. Such technical papers appear to assume that the business rationale for software as a service is self-evident and that the non-technical problems involved in transitioning a company or other organization away from a software as a good towards software as a service are minimal. However, as some authors who take a less purely technical approach observe, the software as a service paradigm has implications for both business and management. For instance, on the engineering management side, Saeed and Jaffar-ur-Rehman [3] note that software as a service has very different engineering requirements than when it is approached as a good (e.g., from the end-user perspective, the service software does not require installation, upgrades, or maintenance). Consequently, they argue that software as a service requires an altered approach to software engineering, and propose a separate Service Based Software Engineering (SBSE) model. This paper builds on their base observation, arguing that the culture and practice of software engineering in a development company is intrinsically tied to the software paradigm in force at the company. Moreover it is argued that building a successful model for services engineering in general will require the examination of low-level engineering (and other) practices throughout an organization and ensuring that they are aligned correctly with the service orientation of the business model.

Eric R. Olsen is currently a part-time lecturer at the School for New Learning at DePaul University, Chicago, IL 60604 USA (e-mail: herolsen@acm.org).

II. Software Paradigm and Engineering Culture

Over a decade ago, Dobson [4] noted that the architecture of any system both embodies and affects the way people think about it. Although Dobson was focused primarily on the technical systems of information technology and telecommunications, his point can be extended to other systems. In our case, his observation can be applied to software engineering. The issue at hand is whether approaching software as a good or as a service affects the practice of software engineering. Conversely, one can also ask how the practice of software engineering affects the business model. Both of these questions are related to the culture of engineering—the way engineering is done on a day to day basis—in a particular organization.

This paper will examine the cultural practices of three aspects of software engineering: planning, versioning, and maintenance. It will do so in the context of a small software company seeking to make the transition from the goods paradigm to the services paradigm.

III. The Data

The observations in this study are based on two sets of data provided by a software development company, call it Company X, with a single product sold into the small and medium-sized business (SMB) market. Company X consists of fifteen employees; nine of those employees are engaged in software engineering tasks ranging from engineering project management, to software design, to software development, to system integration, to quality assurance.

The first set of data consists of an extract from the company's bug tracking system, spanning the eighteen months from September 2004 through March 2006. This period overlaps with the end of a development phase that

1-4244-0318-9/06/$20.00 ©2006 IEEE

resulted in the release of a new version of the company's software (version N.M), the planning for the next version (version N.M+1), and the start of active development on that newer version in March 2005. In this set of data (shown in Table 1), the only data elements accounted for were the fact that a software bug had been reported, when it was reported, and whether or not it was marked as having been resolved by the end of March 2006. The source of the bug report, the effort expended in fixing any particular bug, and the time spent fixing the bug were not considered.

TABLE I
BUGS REPORTED AGAINST VERSION N.M OF SOFTWARE

Month	Total # Bugs Reported this Month	# of Bugs Reported this Month that are Closed at the End of March 2006	# of Bugs from this Month Still Open at End of March 2006	Event
9/04	4	4	0	Integration
10/04	13	13	0	Alpha
11/04	5	4	1	
12/04	19	18	1	
1/05	53	53	0	Beta 1
2/05	58	58	0	
3/05	119	118	1	Beta 2
4/05	73	73	0	
5/05	47	46	1	Release
6/05	54	53	1	
7/05	67	62	5	
8/05	46	40	6	
9/05	34	31	3	
10/05	39	28	11	
11/05	23	11	12	
12/05	49	39	10	
1/06	34	18	16	
2/06	34	4	30	
3/06	44	2	42	

The second set of data is less concrete, consisting of the author's observations at the company over the same time frame. These observations were made informally while working with the engineering group on a variety of software development, build management, and systems integration issues.

IV. DISCUSSION: THE "SOFTWARE AS A GOOD" COMPANY

Until 2005, Company X focused almost exclusively on providing its software as a good. While the sales model was a license with yearly license renewals and support fees, as is typical of much medium and high-end server software, the package sold by Company X was a concrete good in every other way: the software was installed at the customer site on a dedicated server provided by the customer and was managed by the end customer's information technology department.

Aside from providing initial training and help for resolving bugs or configuration problems, once the software was sold, Company X was no longer directly involved. Consequently, it was in Company X's financial interest to encourage its customers to purchase upgrades to the software periodically. Over its decade in business, the company had discovered that most of its customers would tolerate an upgrade every two to three years. Because of this, Company X's engineering efforts were synchronized with this cycle, resulting in a regular, predictable major revision of its software on this schedule.

A. Planning

Software engineering at Company X is a highly methodical process. Each development cycle begins with a planning phase during which the marketing and engineering departments negotiate a set of features to add to the software based on marketing's views of what will attract new sales and engineering's resource constraints. During this time, a meticulous engineering plan for development is assembled, complete with time frames and resource allocations. Once management approves the plan, the development phase begins and each developer works on his assigned projects, reporting regularly on milestones achieved. Towards the end of the development phase, quality assurance begins to do unit testing and report on bugs found. When the various individual development projects are completed, the whole system is integrated and placed into an internal alpha stage where quality assurance attempts to break the software and engineering fixes it as quickly as it can, rebuilding and reintegrating as needed. After this is a two-month beta-1 phase, with the software placed at one or more customer sites for testing; this is followed by a similar beta-2 phase at a different customer site for another two months. During the beta phases, engineering is still involved in bug fixes; however, it is also planning and starting development for the next engineering cycle. Finally, the software is released for new sales and/or upgrades to existing customers.

Note that the central focus of planning is on the sales and upgrade cycle. The only existing customers who are directly involved are those who agree to work on beta-testing; these customers are the first candidates for upgrades. This is planning tailored to the software as a good paradigm where the business model of the company is intent on driving new sales and on selling as many units as possible. While they are important to overall corporate reputation, existing customer relationships are secondary. It is far more important to sell and upgrade to new versions of the software than it is to maintain long-term, stable relationships with the existing customer base.

B. Versioning

While the details of software development at Company X are left to the engineers and their management, the overall direction and pace of engineering is dictated by marketing. As noted above, Company X seeks to synchronize major releases with the upgrade cycle it wants to see in its customer base. To that end, engineering produces a major, new release of software every two to three years. To distinguish this release from other, minor maintenance releases, some component of the software is completely reworked. While this generally improves the software by adding functionality,

increasing performance, or modernizing the software technology used, it also complicates the upgrade process. Instead of requiring a simple patch that applies the changes needed in place and with few problems, upgrades often require disruptive system integration work and restructuring of database schemas. In addition, the changes are significant enough that retraining is generally required for the end-users.

It cannot be claimed that marketing is seeking this disruptive effect on the customers for its own sake. What it is trying to do is to ensure that major releases are seen as big changes, warranting the price tag placed on the upgrade and the disruptions involved in deploying it. In short, marketing is creating the illusion of a new product to justify the repeated resale of what is fundamentally the same good. This means that engineering cycles are not focused on incremental improvements in stability, performance, usability, maintainability, or any other criteria. While these activities do take place, they are overshadowed by the one or two major projects needed by marketing to justify the new version.

C. Maintenance

Some of the patterns mentioned above can also be seen in the maintenance part—the bug fixes—of Company X's software engineering practices. Consider, for example, Figure 1 (developed from the data shown in Table 1).

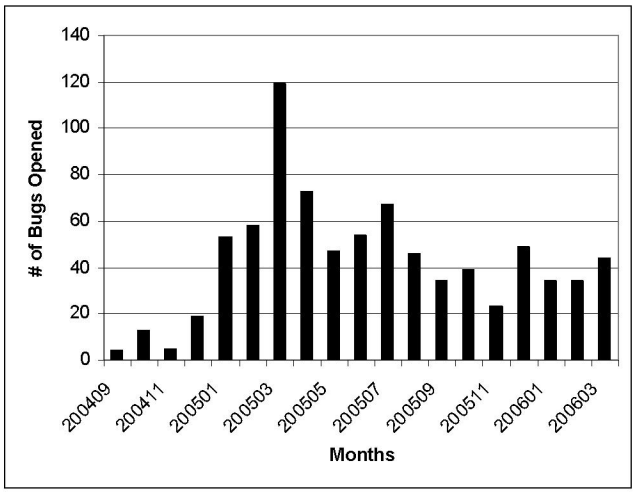

Figure 1. Number of Bugs Opened by Month

This figure shows the number of bugs entered into the Company X bug-tracking system by month. Note the surge in bugs reported in January 2005 (200501), once the beta-1 test period began. Another surge occurred in March 2005 (200503) at the beginning of the beta-2 test period. From that point on, through the release of the software, new bugs reported declined to a fairly steady-state of about 30 bugs per month. Note once again that the severity of bugs is not part of this analysis. These bugs range from spelling errors in the online help system to integration problems with a specific web server.

So far the situation is as expected for normal software engineering practices. Software is complicated stuff and a certain number of bugs will always appear. When a system is first deployed into a "live" environment, more bugs are discovered in the system. As time goes by, the number of new bugs appearing diminishes to a fairly steady-state, but never actually reaches zero.

Now, however, consider Figure 2 (again developed from the data shown in Table 1).

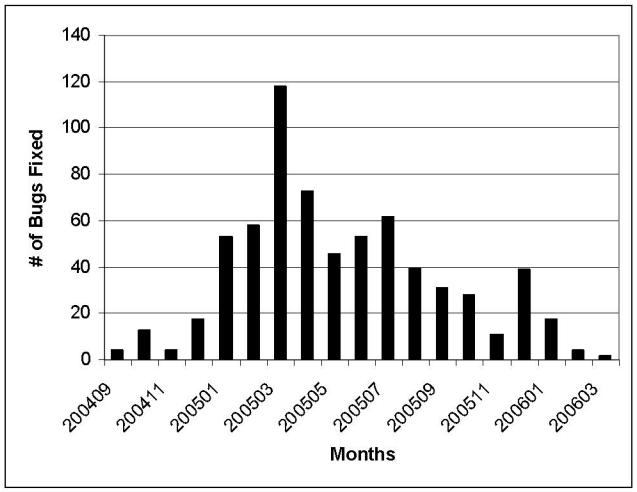

Figure 2. Number of Bugs Closed by Month

This figure shows the number of bugs entered in a given month that have been marked fixed (or otherwise resolved) by the end of March 2006. Note the similarity of the first half of this figure as compared with Figure 1; during this period the majority of the bugs reported have also been fixed. Now note that in the second half of Figure 2, the number of bugs fixed tapers off, approaching zero at the end of time period examined. The tapering-off period begins in July 2005 (200507), two months after the release of version N.M and some time into the development cycle for version N.M+1. This can be better seen in Figure 3 (next page).

What has happened is that engineering resources are no longer focused on support of the already released version, but instead are devoted to new development. The only bugs that are fixed immediately are those that are deemed critical for continued use of the software; anything less stays in the bug-tracking system (known to the engineers at Company X as Nagzilla) as a reminder of what should still be done, but has not yet been completed. These bugs will be fixed if time permits; but time rarely permits. Simply put, software maintenance is not the primary focus of the software as a good paradigm. Resources are better spent on supporting the sales cycle. In fact, the bugs from version N.M remaining in the bug-tracking system after the next release of Company X software will at some point be closed in mass (resolved as "won't fix") once support for that version is officially over.

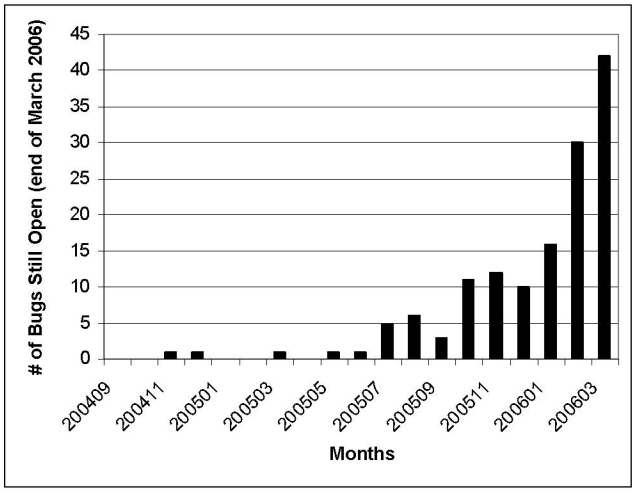

Figure 3. Number of Bugs Still Open at end of March 2006 by Month

V. Discussion: The "Software as a Service" Company

Starting in 2004, Company X decided to switch business models and move into software as a service. Instead of engaging in direct sales and attempting to sell its software to individual clients, it would work with partners who would sell the service provided by Company X's software to their many direct customers. This was seen by management as a way to eliminate the expenses and uncertainties associated with the sales cycle and to concentrate on what the company did best: software development. It also had the merit of accepting the market reality that many of Company X's competitors had done the same thing and were undercutting its prices and generally making the old business model difficult to sustain. By 2005, the company had a major partner (and some smaller partners) which had agreed to offer Company X software as a service from its data centers.

While the business model had transitioned from the goods paradigm to the services paradigm, the culture at Company X had not realigned itself along the new lines. This caused (and continues to cause) a variety of difficulties. These will be discussed below in the context of the engineering practices at Company X.

A. Planning

Software planning at Company X continues to follow a very rigid and formalized software development life-cycle (SDLC). While SDLC itself is not inherently problematic in the services paradigm, it needs to be readjusted for it. For instance, development cycles still extend over many months, as if still coupled to the two to three-year sales and upgrade cycle from the goods company. Since it is no longer necessary to create a distinct version of the software to generate sales, the benefits of long SDLCs are no longer there. In the services environment it would be better to have more frequent, shorter SDLCs in order to respond more quickly to customer needs. For instance, Company X's largest partner would like to have a single-sign-on capability developed across its various service platforms. This is an effort that should take a few weeks at most, but as Company X is tied to a long SDLC that will not produce results for some time.

Another element of the software planning practices at Company X that could be realigned is in modularity. Different software components and the projects to develop them should be kept as isolated from each other as possible (i.e., encapsulated) to make them more easily maintainable and integrated. For instance, Company X recently took on supporting a new version of a database platform required by its major partner as well as a variety of other, smaller projects. Because these were tied together into a single SDLC that had to march in unison toward completion, problems on any one project could stop progress on the others. Due to problems with the database support project, none of the projects were delivered on time—what was the cost to the business in delayed engineering revenue and erosion of reputation due to slippage on more projects than necessary?

With the business model now focused on the services paradigm, Company X would be better off realigning its software practices to fit that model. By doing this it would help foster the relationships with its partners, helping to ensure their strength and longevity. However, there is resistance to doing so in engineering management since "that is not the way we do software" and "quality software requires adherence to the established and proven software engineering practices." Tried and true engineering practices that no longer fit today's needs should be re-engineered.

B. Versioning

One of the major cultural hurdles that Company X has not been able to cross is the idea that it no longer needs to produce version N.M+1, N.M+2, or any other official version of its software. Such versioning is primarily a marketing tactic to convince customers to upgrade and to assure them of the cutting-edge nature of the software.

In fact, Company X does not understand why one minor partner refuses to move to version N.M. The explanation is simple—whatever the new and improved features of this newest released version, it is not worth it to this partner to alarm its existing, paying customer base by disrupting the service provided by the older but proven version of software and returning them to a situation where bugs affected major parts of the software functionality and were found fairly frequently. While this partner happily accepts minor patches to the system, it has no intention of making the move to the newest version without a compelling reason. In fact, when this partner is ready to do so, it will probably survey the entire field of possible service providers and choose the best system available at that time.

This is not to say that Company X should stop developing improved software. However, it does need to rethink the reasons for continuing to do so. New development should not be seen as a marketing and sales activity, but instead as a

relationship building activity. Company X should seek to ensure that its existing partners and clients have no reason to even look at a competitors software. It should focus on ensuring that patches are seamless, that bug fixes are prompt, and that new features integrate like any other patch. The focus should be on providing solid, reliable service.

Furthermore, in providing software as a service, not having an official and public version number attached would provide several benefits. First, the end clients would see only the service, not an ever-changing set of software packages that encourages them to consider packages from other vendors. Second, sales and marketing efforts at the company would lose the ability to distinguish between versions of the product; this would force them to focus on the differences with competitors rather than the differences with version N.M-1. Third, the culture of engineering would be affected by taking away the ability to sweep away old unfixed bugs attached to no-longer supported versions of the software. While this is all fundamentally a psychological management game, it does affect people's behavior.

C. Maintenance

Software maintenance is another area where Company X has not been able to make the cultural transition to the services paradigm. Bug fixes come out on an unpredictable schedule and often require a customer's system to be down for extended periods of time while the engineers figure out why the patch did not work exactly as expected. (This is largely because the controlled engineering practices are there for new development, but not for deployment of fixes.) Worse, bugs are seen by the engineers as a mostly unpleasant chore that takes away from their time doing interesting new development.

In the service paradigm, software maintenance becomes one of the most important activities in software engineering. Since software is not built as a good to be used and then thrown away when something newer and better comes along, it has to be kept up. Once a new software system had stabilized (i.e., had reached the point at which most bugs are of minor impact to the majority of customers and had reduced to a steady trickle of new reports), software maintenance in this paradigm would have two activities that should be virtually indistinguishable to the end customer: bug fixes and enhancements.

In order to maintain the best relationships with end customers, bug fixes cannot be allowed to languish in the bug system, only to be eventually forgotten and abandoned. Instead, engineering resources should be devoted to ensuring that all bugs are resolved in a satisfactory and timely fashion. This is part of providing the best service possible. Resolving bugs of any sort needs to be seen as a central engineering activity, not a chore to be done when time permits.

Enhancements should be treated as bug fixes in that they should come in self-contained, non-disruptive patches. By containing each enhancement in an independent, small-scale SDLC, it can be developed, tested, and integrated on a similarly small scale. If the software is well-designed to be modular, most enhancement should have minimal side-effects in other components. This should make deploying them non-disruptive, almost invisible to the end-user. More extensive enhancements could require more than a back-end patch. However, in a service environment, it is in the company's best interest to put resource into making sure that deploying the enhancement has little impact on the end user.

In all of these cases, the company needs to keep in mind that the purchaser of its services only wants the services. It does not want to deal with disruptions to service. As the market for software as a service develops, this will be one of the distinguishing features by which services are selected.

VI. Conclusions

As described in the discussion above, management at Company X has decided to move the company towards the service paradigm and away from the goods paradigm. While they have done this on the business side in their partnerships and in eliminating their direct sales efforts, they have not followed the change of model into the rest of the company, engineering in particular. Simply put, software as a good is different from software as a service. Furthermore, the practices and culture of much traditional software engineering have been developed in the goods environment and need to be realigned for the services environment.

On a more general level, as a practice of service engineering develops, researchers will need to pay close attention to the small details of engineering practice and culture. They should ask themselves questions like: How should we alter our engineering management practices to support our business model? What effect does calling a software package version "3.4" have on the business and on engineering? Why do some bugs get fixed immediately and others remain unexamined for months? If corporate practices at all levels can be properly aligned with the business model chosen, a company can make itself more fit to compete in the marketplace and be sure to place resources where they are actually needed.

Acknowledgment

The author wishes to thank Dr. Robin Qiu of Pennsylvania State University for his encouragement in writing this paper and for his early comments which helped focus and make this paper more coherent.

References

[1] H. Guan , B. Jin, J. Wei, W. Xu, N. Chen, "A framework for application server based web services management," in *Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC '05)*, 15-17 Dec 2005, pp. 95-102.

[2] M. Turner, D. Budgen, P. Brereton, "Turning software into a service," *Computer*, vol. 36, no. 10, pp. 38-44, Oct 2003.

[3] M. Saeed, M. Jaffar-Ur-Rehman, "Enhancement of software engineering by shifting from software product to software service," in *Proc. of the First International Conference on Information and Communication Technologies (ICICT 2005)*, 27-28 Aug 2005, pp. 302-308.

[4] J. Dobson, "Issues for service engineering," in *Proc. First International Workshop on Services in Distributed and Networked Environments*, Prague, Czech Republic, 27-28 Jun 1994, pp. 4-10.