A Case Study of Post-Deployment User Feedback Triage

Andrew J. Ko, Michael J. Lee, Valentina Ferrari, Steven Ip, and Charlie Tran
The Information School | DUB Group | University of Washington
{ajko, mjslee, ferrariv, iperton, ctran7}@uw.edu

ABSTRACT

Many software requirements are identified only after a product is deployed, once users have had a chance to try the software and provide feedback. Unfortunately, addressing such feedback is not always straightforward, even when a team is fully invested in user-centered design. To investigate what constrains a teams evolution decisions, we performed a 6-month field study of a team employing iterative user-centered design methods to the design, deployment and evolution of a web application for a university community. Across interviews with the team, analyses of their bug reports, and further interviews with both users and non-adopters of the application, we found most of the constraints on addressing user feedback emerged from conflicts between users heterogeneous use of information and inflexible assumptions in the team's software architecture derived from earlier user research. These findings highlight the need for new approaches to expressing and validating assumptions from user research as software evolves.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Corrections, enhancements, extensibility.*

General Terms

Human Factors, Design, Management.

Keywords

User feedback, bug reports, bug triage, software evolution.

1. INTRODUCTION

Designers rarely know everything about user needs before a product ships. Stakeholders are overlooked [16], use cases are missed [20] and above all else, the world changes, requiring software teams to evolve applications to suit new needs. It is therefore inevitable that much of the work to serve user needs through design happens *after* software is deployed, in continuously changing contexts of use [24].

But as most practitioners in the software industry know, changing software is not so simple. For example, software engineering researchers have long studied notions of coupling and cohesion [25], modularity, and cross-cutting concerns [9], analyzing the role of technical dependencies in both constraining and facilitating change. Moreover, there are several economic [2] and lifecycle [19] factors that can limit software change, not to mention a variety of cognitive [8] and social [18] challenges in simply understanding complex software systems in order to change them.

One aspect of software evolution that has received little attention, however, is the role of post-deployment user feedback such as support requests and bug reports. With the rise of web-based technical support and the ease with which users can contact small software teams via e-mail and the web, what constrains a software team's ability to address user feedback with software changes, even when a team is committed to user-centered, iterative design?

To investigate this question, we performed a 6-month field study of a software team employing Agile methods and staffing several user researchers and designers working directly with developers, testers, and managers. We report on the history of one of the team's products, a grade book application for a university community. We discuss the team's user research, prototyping and post-deployment iteration, analyzing the constraints they faced in addressing post-deployment user feedback. We also analyzed over 1,200 bug reports the team did and did not address and the reasons why; we also interviewed a sample of both users and non-adopters of the team's application, revealing needs the system did not serve and what constraints prevented the team from serving them.

Our findings make several contributions to knowledge about user-centered design and software evolution. In particular, we found that most of the constraints in addressing user feedback emerged from conflicts between (1) heterogeneous perspectives on how grades should be represented and (2) global assumptions in the team's software architecture and user interface design. When the team attempted to address these conflicting user needs, the resulting solutions were considered inadequate by both the team and the user community, limiting changes to incremental modifications that supported existing users. These findings highlight the need for new approaches to expressing and validating assumptions from user research as a team receives and triages user feedback.

In the rest of this paper, we discuss prior work on software evolution and then detail the methods used to study the team. We then discuss our observations and their implications on user research, user-centered design and software evolution.

2. RELATED WORK

We know of know prior work that has explicitly investigated the constraints that software teams face in addressing software change requests. There is, however, considerable prior work on the factors that can constrain software evolution in general, ranging from the inflexibility of computer code, the time required to invest in change, and the skills available to implement change, to more systemic factors such as policy, market forces, and politics. In this section, we discuss prior work on these various factors.

One major constraint on software change is *complexity*. For example, Buxton argues that as systems grow in complexity, the architecture, technologies and paradigms "create a straightjacket that severely affects the cost of change." [6]. Lehman provided one of the first reports on this phenomenon [19], deriving several laws of software evolution from a study of several long-lived applications. Lehman argued that because of the ease with which code can be reused, there is an incentive to implement changes with existing code, rather than aggregate changes into new code.

These forms of reuse are captured in several concepts of code complexity, such as *coupling* (the degree to which program modules are mutually interdependent) [25] and *cross-cutting*

CHASE'11, May 21, 2011, Waikiki, Honolulu, HI, USA

concerns [9] (the degree to which a software feature spans modules). Measures of these concepts across successive versions of software show that all tend to increase, causing each change to span a larger subset of a program's modules [21][7]. One way that teams mitigate increases in complexity is through *change impact analysis* [1], determining which parts of a program will be affected if a proposed change is made. Recent studies have shown that when developers cannot find enough information to assess the impact of a change, the risk of the change introducing new defects, breaking an existing use, or otherwise changing user experience, is assumed to be prohibitively high [18].

Another constraint on software change can be the user community itself. For example, Buxton argues that because users have made an investment in learning the product; any significant changes to the UI or workflow may threaten the loss of existing users and may not lead to new adopters. After all, learning is a significant investment [14] and some users may see no value in the new benefits offered by a system relative to the new cost of learning the system [2]. Moreover, people often adopt new technologies not on the *actual* cost, risks, and rewards, but *perceived* ones [5]. People are also adept at appropriating software in unexpected and idiosyncratic ways [15], leading them to depend on code in ways a team may not have intended. Once these use dependencies are established, modifying such code may mean breaking unplanned but widespread uses, even if the code was viewed as provisional.

Teams also face infrastructural constraints. For example, Edwards et al. [10] explain how infrastructure can preclude certain user experiences, expose technical abstractions to users in undesirable ways, and force users to interact directly with infrastructure to accomplish their goals. Such infrastructural constraints can also be a significant constraint for software teams' ability to change software, forcing them to select undesirable designs because they are not free to change the infrastructure.

There may also be social and cognitive factors within a team that constrain change. For example, teams may experience *loss aversion*, strongly preferring avoiding losses to acquiring gains [17]; with respect to software evolution, this may mean avoiding losing a small number of existing users over gaining a large number of new users, or avoiding an architectural change even though it may enable significant improvement in user experience. Similarly, teams may engage in *irrational escalation* [4], justifying increased investment in a decision because of cumulative prior investment, despite new evidence that the decision may be ineffective. Teams may also experience *confirmation bias* [4], seeking or interpreting information in a way that confirms preconceptions about how software is used or what value users derive from it.

3. METHOD

The focus of our study was on understanding the role of user feedback in addressing software change requests. In this section, we describe the team we observed and how we analyzed their efforts.

In selecting a software team for study, we sought one that had an explicit focus on user-centered design. We chose to study an in-house software team at a university known as LST, consisting of 20 full time and 40 part time staff. While the team was local, making it easy for us to observe the team, the primarily reason we selected the team was their mission statement:

> "We follow an iterative, user-centered design and development process that focuses on understanding the needs and experiences of our users. Whether we are creating a new tool or updating an older one, our design decisions are based on direct feedback, user research, and findings from usability studies."

This dedication was reflected in their many awards, with the most recent presented by the Center for Digital Government and Education, and the ACM Special Interest Group on University and College Computing Services. This was also reflected in their actual work throughout our observations.

The focus of our study was on LST's most recent tool, a cloud-based web application called GradeBook, used by university instructional staff to store, organize, and publish student grades. The main screen of this application appears in Figure 1. The major features of the application include a spreadsheet-like interface for storing scores on assignments, categories for representing groups of assignments, automatic final grade calculation, and online grade submission. The application also imports and exports Excel files. We discuss the rationale behind several of GradeBook's features later in our discussion of the team's initial user research.

The GradeBook design team included six individuals across two teams: one focused on the grade book application itself and another focused on online grade submission. Of these 6 individuals, 2 were developers (and participated on both teams), 3 were designers (one of which participated on both teams), and 1 was the program manager for both teams. One of the designers focused on client-side user interface design, writing HTML, CSS, and JavaScript for the front end. The two developers were responsible for the majority of the engineering work behind all versions of the application.

To learn about the project history, we performed two semi-structured interviews with the 4 of the 6 team members. We interviewed them in pairs to help reveal conflicting and confirming memories about the project history, also asking the same questions in different ways to cross-validate responses. Our questions focused on several aspects of GradeBook's history: we asked about the rationale for the project's inception, what user research was performed to inform the design, and about the results of the user research. We asked about the rationale for the major features of the GradeBook application and their relation to the user research. We also asked what aspects of GradeBook had evolved in response to user feedback, and which aspects the team wanted to evolve, but could not and why. Interviews were recorded and transcribed.

In addition to interviewing the team, we also analyzed the team's technical support and bug report repositories. The team used Bugzilla to track issues and classified resolved bugs into FIXED, WONTFIX, LATER, DUPLICATE, NOTREPRODUCIBLE, MOVED, or REMIND. Our analyses focused on the 1,046 FIXED and 144 WONTFIX closed reports. In reading them, we focused on understanding what aspect of GradeBook was identified and why the team decided to fix or not fix the issue. It was common for reports to both indicate the rationale for closing a report, a link to the code change in the version repository (if there was one), and a link to support tickets that prompted the project, if any.

To understand the user community's use (and non-adoption) of GradeBook, we also interviewed several instructional staff in charge of teaching the large undergraduate population, particularly those teaching introductory lower-division courses. We focused on

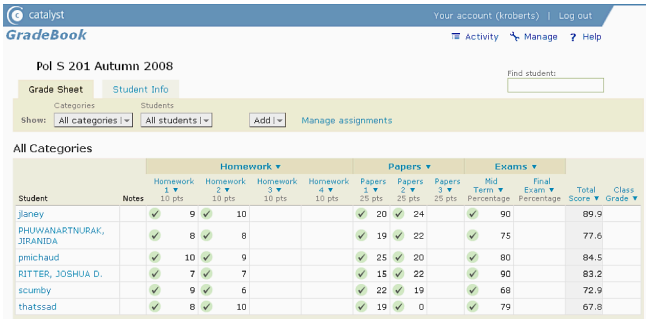

Figure 1. The GradeBook application designed by LST, as originally released. Data in the spreadsheet is fictional.

departments with more than 100 students and on courses being taught during the quarters of our observations. We contacted each of the instructors and teaching assistants of these classes through email, explaining our study and asking for participation. Of the 82 instructors and teaching assistants we contacted, 22 replied. Of these replies, we successfully arranged interviews with staff of the 12 courses in Table 1. Of the 12 staff interviewed, 5 were the official instructors of record, 6 were teaching assistants, and 1 was a course coordinator, responsible for managing teaching assistants in collaboration with the instructor. Of the 12, all used Excel to track some form of student grades, 6 used GradeBook to store grades, and 2 used other grade management software mandated by their departments. All used GradeBook to submit grades online.

Our semi-structured interviews with these 12 instructional staff involved a walkthrough of the syllabus and rationale, the kinds of deliverables students submitted, how deliverables were submitted, how they were graded, where grades were stored, how feedback and scores were provided to students, the tools used for all of these processes and the staffs' views on these tools' inadequacies for grading. Each interview was audio-recorded and transcribed.

4. SOFTWARE DESIGN PROCESS

In this section, we describe the team's user research and prototyping for GradeBook, and the basic elements of the design they initially deployed. We use [dev], [pm], and [des] to refer to quotes from developers, the program manager, and designers, respectively.

Prior to working on GradeBook, the team had a 7 year history of creating other web-based applications This influenced the design philosophy behind GradeBook:

> [pm] ... we started out in 1998 with WebQ [a quiz application] as the first tool. And so we kind of grew the toolset over time by building new things.... And yet we knew all along that the Catalyst tools are valuable and useful in a way that courseware like Blackboard isn't, precisely because they are modular and a faculty member can ... use them in contexts and ways that aren't course-centric and locked down. So it has been organic, but also strategic.

In early 2007, the team began hearing from the community the need to move beyond paper grade submission:

> [dev] ... when I was first here in 2005 we said we weren't going to because it would be course-centric, then we kind of moved into this space where lots of people were saying, "why do I have to fill in this bubble sheet?," and it just felt like, to us, that, our group had the right skill set to make a course tool like GradeBook ... there wasn't anything out there that was easily integratable with campus infrastructure...

In November 2007, the team began user research. While no user research is entirely comprehensive, we found the team's efforts substantial, triangulating research from interviews, surveys, and artifacts and collaboration with domain experts. In particular, a major part of the research was working with others in the university community who had experience developing custom grade management software for particular departments, particularly the computer science department:

> [dev] ...we interviewed some of those CSE folks, and we worked with the developer, [name omitted], 'cause he had a lot of knowledge on what the requirements were for [the CS grade repository].

The team's primary research efforts were 2 to 3 months of interviews and surveys with instructional staff and students.

> [dev] we did a survey of faculty, TAs, people who did on the administrative side, grading sort of stuff... I think we maybe interviewed 7 or 8 sort of faculty, people who were actively teaching and recording grades. And then of course other people who were developers and administrative types. That's where most of our requirements came from.

To recruit these individuals, the team used snowball sampling, starting with existing contacts who taught online courses, as well as members of the community who had previously volunteered for surveys, interviews, focus groups, and usability tests.

course	# contacted	# replied	who was interviewed
Japanese (JAPAN)	2	2	1 instructor
Spanish (SPAN)	7	2	1 teaching assistant
Chemistry (CHEM)	3	2	1 teaching assistant
Biology (BIO)	2	2	1 course coordinator
Mathematics (MATH)	7	2	1 instructor
Computer Science (CSE)	1	1	1 instructor
Electrical Engr. (EE)	1	1	1 instructor
Mechanical Engr. (ME)	1	1	1 instructor
Music (MUSIC)	2	2	1 teaching assistant
Economics (ECON)	16	2	1 teaching assistant
Accounting (ACCNT)	4	2	1 teaching assistant
Communications (COM)	6	1	1 teaching assistant

Table 1. The 12 courses for which interviews were conducted. Columns indicate how many staff were contacted, how many replied, and the role of the individual interviewed.

In interviews, the team found that most faculty used Excel, coordinating with TAs with e-mail attachments, so they gathered a large collection of Excel spreadsheets from faculty, creating a repository that was used to examine the range of ways that faculty stored, organized, weighed, and ultimately computed final 4.0 grade points. The team found that most teachers organized deliverables into categories of assignments (for example, a course might consist of multiple exams, quizzes, assignments, etc.). The team also found it was common for each of these different categories to have different grading scales; some would be pass/fail, others would be based on percentages, and others still might be based on points. While this did not account for all of the uses they observed in spreadsheets, it covered most.

The team also surveyed the instructional staff in the community, finding that the most important desired features were being able to weigh assignments, coordinate grading work with TAs online, adjust grades, and provide feedback about grades to students. The team deployed a similar survey to the student body, who indicated that the most important features were tracking their progress on grades and understanding how their grade was calculated.

After several months of data collection and analysis, the team completed the research in winter 2008 and began a six month phase of design and implementation. They began by prototyping a simple mockup in order to solicit feedback from instructional staff:

> [dev] I think we spent a lot more time than we normally do in our initial designs. So after we were done with all of our initial research, I think we had some initial screenshots, we did a rapid prototyping thing, it was one of the first times I think we'd really done that... Just to get something really quick and dirty for people to play with.

After recruiting staff to use the prototype in a range of usability and feasibility tests, the team ultimately arrived at an idea for a cloud-based spreadsheet, mimicking Google Spreadsheets, creating a single data store for course grades that faculty and TAs could all access from web browsers. They focused on designing a flexible platform for addressing post-deployment user feedback:

> [dev] ... for the initial release we just needed it to be very generic, so we could do lots of neat detailed stuff... The first goal was just to get it so that people could create assignments, add grades, publish those grades to their students...

As seen in Figure 1, the core feature was a grid of students and assignments. Each row stored a student, notes about the student, and a collection of assignment categories. Four assignment scoring scales were supported, including a *point*, *percentage*, *text* and *custom* scale (which allowed instructors to define a mapping from ordinal text values to percentages). Assignment scores could also be published to students online. Total grades could also be calculated automatically, based on a weighted sum of each category's assignments (with the option to drop 1 or more assignments from a

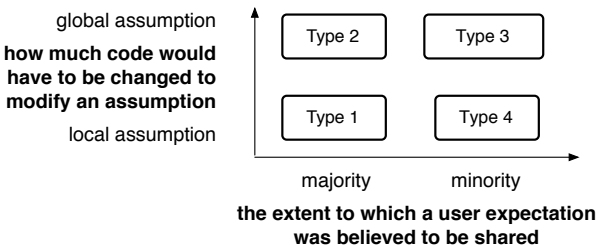

Figure 2. Two constraining factors that explained many of the team's software evolution decisions.

category). The GradeBook UI provided several ways of filtering the spreadsheet view to specific assignments. It also provided a preview mode for instructors to view students' view of published grades.

5. FOUR TYPES OF SOFTWARE CHANGE

Our interactions with the team began approximately 1 year after the team's initial launch in September 2008 and continued for six months. Our focus in analyzing the data was understanding what factors best explained which change requests the team did and did not address. We analyzed these changes by considering the team's rationale statements from the interviews and bug reports, identifying each rationale statement and inductively arriving at a set of *constraining factors* to which the team appealed in justifying their triage decisions on software change requests.

While the team encountered many of the constraints discussed in our prior work (competition with other grade book software, campus politics, modularity challenges), there were two factors that the team appealed to throughout both the interviews and bug reports. One of these factors was *how much code would have to be changed to modify an assumption* (Figure 2's y-axis). Some changes were local: for example, the team received requests to validate assignment scores in ways that were explicitly incompatible with the existing validation. Other assumptions were more global: for example, most of GradeBook's code assumed that assignments have a single score and changing this assumption would have required a major rewrite.

The other constraining factor affecting the team's decisions was the *extent to which the user expectation motivating a proposed change was believed to be shared* by the larger user population (the x-axis in Figure 2). The team rarely had enough data from user research to estimate the extent to which such expectations were shared, often resorting to using their instincts for such estimates or excluding particular uses from scope to avoid needing to support it. Moreover, many of the user expectations were ill-defined not only in the team's mind, but in the mind's of the users' they spoke to, meaning that prototyping new features to support these expectations caused expectations to shift, once users were able to work with a computational version of what they had in their minds.

All but the Type 1 changes in Figure 2 were difficult for the team to address. When the team pursued them anyway, they required significant effort either in re-implementing features or fielding new user feedback; moreover, these changes generally led to results that were unsatisfactory to both the team and the user community. In the rest of this section, we describe several examples of these changes, illustrating how the two factors in Figure 2 interacted to constrain how the team could respond to post-deployment user feedback.

In quoting from bug reports, we use [wontfix] and [fixed] to represent reports. Underlined text is from the bug report description; other text represents text from bug report comments. For interviews with instructors, we use the disciplinary abbreviations that appear in Table 1 (e.g., [CSE] represents a quote from the CSE instructor).

5.1. Majority Expectations, Local Assumptions

Type 1 changes concerned expectations that the team believed were largely shared by the population and were highly localized in GradeBook's implementation. These issues are best characterized as *bugs*: once reported, they were both straightforward to address and desired by both users and the team. To analyze these, we focused on the 1,046 FIXED bug reports that the team had filed between GradeBook's launch and the end of our observations, analyzing the changes that modified a one or two files and had little to no discussion in reports about how the change should be implemented.

We found that most of these changes were straightforward because they changed the *parameters* of behaviors that were already explicitly or easily parameterized. For example, most of these changes modified labels, images, colors, links, and layout in the user interface. These also included changes to interactive behaviors, such as whether a dialog was modal. Existing behaviors were also made *conditional*, for example, accounting for overlooked error cases, excluding data that was already computed, or validating data before accepting it. Although the team did not have explicit evidence that these changes were desired by users, most changes concerned violations of consistency and convention, and so the team rarely discussed whether to make them.

5.2. Majority Expectations, Global Assumptions

Unlike the Type 1 changes in the previous section, the Type 2 changes emerging from post-deployment user feedback conflicted directly with assumptions made in GradeBook's implementation. In this section, we present two such changes.

Equating Groups and Class Lists. A major source of feedback early after GradeBook's initial release regarded the confusing workflow for giving privileges to other staff to view a GradeBook:

> [dev] From the users perspective, before you had to go in and say "I have a class list created for it, but I don't have a group. So I need to create a new group and from within the group I need to attach the class list to the group and then go back to the tool and then attach the group that has the class list."

The team's research showed that most users viewed *groups* and *class lists* as equivalent; this shared expectation, however, was in conflict with the global assumption in all of team's other applications that a group was a different thing from a class list. The team carefully considered whether the change was really necessary, initially deciding to hide the group complexity behind the UI:

> [dev] ... we didn't want to change all of our backend... it was going to take too much time and it was too hard... we're just going to do this half way...

As this global assumption became more problematic in other applications, the team eventually decided that groups and class lists needed to be equated. The data migration efforts were substantial:

> [dev] ... what we had to do was instead of just saying okay here's a new group ID, we had to remove the old one, and since group A might have contained two course groups as well as some people, it then became three groups, so then we had to say, okay GradeBook, you're actually now going to use three groups, instead of just one. That was a huge data migration process and it caused a lot of pain, and that was probably like 3, 4 months of time.

Moreover, once the migration was complete, the team needed to undo the hacks that make GradeBook's assumptions consistent with the newly obsolete data model:

> [dev] We ended up paying for it later on when we had to undo that work... when we made this group change, in addition to the migration, we had to go in in GradeBook and change all the code that was making that assumption for us, and remove the ad hoc group from ever being created, because we didn't need it anymore, because you could just add the groups directly.

More than a year later, the developer was still discovering places in the implementation that made the old assumptions:

> [dev] Well, in this case, I mean I think I fixed a bug around this redo, two weeks ago still? So, it's one of those things that tend to keep cropping up, because you have all this code that depends on these few assumptions that you'd made and then you change it in 90% of the places, and if you miss any of them, no matter how hard you look (I find that I always miss some), it always comes back.

WebQ and CollectIt Integration. While the previous example involved a change *within* GradeBook, the team also pursued changes *between* applications. For example, the team often received requests to integrate with other Catalyst tools, particularly WebQ (which allowed faculty to create scored quizzes for classes) and CollectIt (which allowed instructors to create digital drop boxes for assignment submissions).

The team's first integration was with WebQ; as with the change in the previous section, the challenges stemmed from incompatibilities in the GradeBook and WebQ data models:

> [dev] ... because some of the flexibility that WebQ gives you in grading the quiz, you can allow students to take it twice, so you might have two different grades. GradeBook doesn't really account for that, it wasn't really made for that... And you can also have branching; ... its possible for some students to have a quiz out of 40 points and another student might have, which is the same quiz, out of 20 points, you know. GradeBook's not set up for that.

> [pm] Import was really hard. ... They each have their own data stores and their own interface ... there are just all these little things that have to be checked and user confirmed, so it's pretty awkward.

Therefore, while the team ultimately succeeded at implementing a solution, the conflicts in the data models were necessarily exposed in the GradeBook user interface. Any further changes the team desired, particularly that of importing multiple values for a single quiz, were constrained by decisions made in the initial version of the import process. For example, the team considered supporting multiple values for GradeBook assignments:

> [dev] ... basically we'd have to do a data port, because you have all these data entries with one single value, you'd have to do something like enter, you know, a linker table that's pointing off to a series of values, or somehow change that so you could perceive multiple values... the real challenge there would be that you've already released this to the public, so you need to make sure that existing things still work when you change the data.

The WebQ integration experience made the team more hesitant to move forward with CollectIt integration.

> [pm] I mean we've been wanting to allow people to have CollectIt scores import. And the problem there is that CollectIt doesn't have *any* concept of a score. It has a, "let's have a conversation" feedback, there's no idea of giving a point or any kind of scale or anything there.

The GradeBook developers explored several alternatives to these data integration challenges, but faced a tradeoff between simplifying import and supporting flexibility:

> [des2] We actually worked on some whiteboarding sessions on integrating CollectIt with GradeBook, and we thought we had something that made sense... Even the thing that you and I came up with, which made a lot of sense to me, and took care of a lot of the edge casey type of stuff, we weren't really sure if that was sort of a model people would understand... Sort of that dial of easy versus advanced and flexibility versus rigidity.

Ultimately, the team felt that in all of these integration efforts between the existing systems, they were limited in their ability to offer straightforward, usable interactions. With WebQ, the import process was necessarily complex because of the data scheme differences, and with CollectIt, there would have been significant changes to how GradeBook represents scores on assignments. In both cases, the team felt the tradeoffs might not lead to a net improvement in user experience. The program manager believed that these tradeoffs were problematic enough that data integration and rewrite was the only solution to simplifying the workflow:

> [pm] We're going to have to go the other way and enable much more integration and make different kinds of data available at the surface... Its so deep down in the data store that its not even possible…

5.3. Minority Expectations, Global Assumptions

Like the Type 2 cases we described, the Type 3 changes conflicted with expectations that spanned GradeBook's implementation. In contrast, however, Type 3 changes concerned user expectations that the team perceived to be idiosyncratic, but severe enough to be addressed. We present two such changes, each receiving some attention from the team, but leading to changes that were ultimately constrained by assumptions in GradeBook's implementation.

Improving UI Performance for Large Grids. One of the major assumptions the team made in their initial testing was about the number of students and assignments each GradeBook would have to maintain; the team tested courses with several dozens of students and 10-20 assignments, because those were the ranges encountered in earlier user research. In developing GradeBook, these performance profile assumptions became quite global, reaching into the user interface, the server interactions, and the event-handling mechanisms that coordinated the two.

Post-deployment, however, the team quickly realized that some of GradeBooks' student and assignment counts far exceeded these tested limits. For example, one problem was with initial loading, requiring a significant rewrite:

> [des2] ...performance kind of depended on the number of students that you had in your class. So people with really large classes, now we don't load everybody up right when you load the screen... its in a big table, it depends on the browser you're using.

While the team became increasingly aware of the performance issues through its testing, their approach to reacting to user feedback about Type 3 changes was passive, waiting for explicit complaints from users. For example, one of the team's testers reported the problem in a bug report:

> [wontfix] IE choppy when scrolling through large class lists in FGR — For a class of about 200 students IE7 has a difficult time handling the FGR. Using the scroll wheel or arrows is typically choppy...

A developer on the project closed the report, arguing:

> [wontfix] we went through spring quarter with zero complaints of choppiness... closing this bug.

Six months later, one of the instructors of the introductory computer science courses wrote in:

> [fixed] I just don't think Catalyst Gradebook is practical to use as a web app for large courses... When I try to look at a student's grade, I scroll or page down the worksheet, and it seems to load the students 5 at a time with Ajax... It can literally take 2-3 minutes just to find a student in the giant page while all the kids are loading.

The team responded in several ways with performance improvements, even testing the changes on example GradeBooks with 300-600 people and 10 assignments, tuning performance for larger classes. Ultimately, the developers were not satisfied:

> [dev] In my opinion there is no 'good' fix for this. Either we slow down the initial page load, or we do scroll as you load and lose the context...

Another major performance problem was caused by the number of assignments some teachers tracked in GradeBook:

> [pm] ... if you look at some people's GradeBooks, they have so many columns because they're tracking daily participation... that would be difficult to change because we decided to put a grid view.

To change the UI from a grid view fit to a browser's width to a view that could scroll was infeasible for a number of reasons. The grid view was based on a 3rd-party library, which did not support such a view; moreover, most of the application's UI code was built on the assumption that the grid was always the width of the window.

These performance problems were a critical concern in our interviews with both users and non-users of GradeBook:

> [CSE] Gradebook is not good at handling a course that has 700 students, at least the last time that was the case. So when I do import everything at the end, just for that brief moment like we talked about, it's, I pray that nothing will be wrong... the entire page becomes really choppy because there's so many people.

> [EE] ... by the end of the quarter it's very slow, for some reason I enter a grade, I lose a lot of them and I have to go back and fix them later... it has added another layer of responsibility to instructors that already hurts a workload that's too high.

Although the users in our interviews viewed the performance problems as critical, only the computer science instructor had actually reported the problems to Catalyst. In fact, most interviewees were excited that we had interviewed them, because they expected us to report their feedback to the team. And yet,

because the team waited passively for feedback, they were not aware of the significance of the performance problems until they it was too late to address them. This, in the team's view, greatly affected GradeBook's adoption:

> [dev] Unfortunately, I think a tool gets released, they check it out, and then they go, oh, its too slow. Okay, well we hear that and we fix it, but if your first impression of the tool is that its too slow, its not a whole lot to bring you back the second and third time.

Variations on Extra Credit. Another assumption underlying GradeBook's implementation was the weighted sum and dropped scores approach to computing final grades. The team knew that there were exceptions to this approach, particularly with respect to extra credit, but they did not account for them in the initial design. This became a frequent topic of user feedback:

> [pm] We found in our user research that a lot of faculty use extra credit but there wasn't any consistent pattern. The one thing that we did to support that was you can add more points than are possible, so you can have an assignment worth a hundred points and give people a hundred five. But that doesn't work for a lot of people. What a lot of people want to do is have a whole extra credit assignment that gets added on as extra in the category weighting... Its a big change and we hear that request a lot, and what we usually do is to help people download their scores and do the calculation in Excel...

The team proposed similar workarounds to users desiring other alternatives to the weighted sum model. For example, many faculty asked for explicit support for various types of class curves; in most cases, the team suggested falling back to Excel. These workarounds represented one way to escape the assumptions underlying GradeBook, while still finding a way to support users' alternative practices, but the impact on users' workflow was inevitable. Half of the users we interviewed said that the lack of support for these practices was a primary reason for using Excel instead.

5.4. Minority Expectations, Local Assumptions

In contrast to the previous three types of changes, Type 4 changes were primarily constrained by the *variation* in user expectations perceived by the team. The team did not see obvious ways to express these heterogeneous and often conflicting expectations in a way that would preserve GradeBook's simplicity. In this section, we present three such desired changes, showing how the team ultimately defaulted to the assumptions already expressed in code.

Exceptional Meanings to Assignment Values. One class of post-deployment feedback regarded how GradeBook handled the assignment scores. Many of the assumptions built into score validation were incompatible with some users' practices, particularly in computing final grades. These incompatibilities forced the team to predict which of two user expectations—the implemented one or the one reported in user feedback—was more commonly desired. One example of this was in the meaning of particular grades. For example, in one case a user pointed out that "X" was a valid grade, but when importing an Excel spreadsheet with an X grade, GradeBook marked it as invalid until the user explicitly selects "X - No grade now" (the GradeBook equivalent). The user wanted the conversion to be automatic, but the developer argued that this was not a safe assumption:

> [wontfix] I think this concern is bogus (to be pedantic X - No grade now is not even a grade), and transforming a 'X' to 'X - No grade now' seems like a big leap to me... We want because we want to be sure he's gone through them and specifically assigned an "X" or an "I" and that it isn't some mistake. The other factor that is causing this is that he is not really a GradeBook user, but someone trying to import grades at the end of the quarter for the sole purpose of submitting...

The developer's rationale for not making this change stemmed both from a prediction that most users would rather know about data entry errors than save time, and from a belief that the user was not "not really a gradebook user." The team's reluctance to support exceptional meanings of values was characterized well by the team's manager:

> [pm] The mantra that we started using to help us decide what features are in, what features are out is, we're not Excel... we were trying to make an online GradeBook that was useful, but didn't go into a lot of calculation and fine tuning, especially around the issues where there was not wide agreement about how things were done.

The team found that this mantra was important in communicating to users *why* different interpretations could not be supported:

> [des1] I say it to users; "we can't rebuild Excel" and that resonates with them. They say, "Yeah, I guess that's true."

The 4.0 Assignment Grading Scale. In the previous case, users identified needs that conflicted with assumptions made in GradeBook's implementation. In this next case, however, the needs themselves, as expressed by instructors, were quite homogenous; it was the *reactions* to the team's expression of those needs in code that varied. Originally, GradeBook supported a small set of basic grading scales. However, post deployment, the team received feedback about the desire for a scale that matched the 4.0 grade point scale used in final course grades:

> [dev] Initially we said we're not going to do a grade point scale, we're going to do something more broadly usable. And that's when we came up with these custom labels that I was describing earlier. And people were like, I want my grade point scale, I want my grade point scale, and so we had to have a grade point scale.

According to the developers, actually implementing variations of the 4.0 scale was straightforward:

> [dev] ... it was an additive change. We were already supporting like 3 or 4 scales and so we added that one. And that was just a table addition. We didn't have to actually migrate any data or anything... the only work was some custom JavaScript and there was no, there was very little back end changes that needed to happen.

The actual interpretation of what users *meant* by a 4.0 scale, was an entirely different problem. As the program manager described, the way that 4.0 scales actually being used by faculty were not amenable to formalization:

> [pm] We didn't initially support 4.0 scale scores. And this has been, its really a pedagogical debate, in some ways... A lot of faculty want to use 4.0 scale grades for all assignments in their class and then do calculation on those. And the software says, "those aren't actually real numbers, those are more like a ranking," because its not a literal scale from 0 to 4. But trying to communicate to faculty who've been doing this for years in Excel and thinking there's absolutely nothing wrong with it is really difficult.

Their initial efforts to design a feature to fit the practices they observed in user research were unworkable:

> [pm] ... we kept saying, this doesn't make any sense, this doesn't make any sense, this is really hard to use compared to the class grade, and then we just sort of scrapped it all and started over a few months later... we got to the end where we sort of had a Frankenstein, where it was doing it one way here and another way there. We thought for a while and we said, wait a second, we can't release this.

The team ultimately arrived at a solution that represented a compromise between many conflicting views on the meaning of scale. However, this inevitably led to feedback from users whose practices conflicted. For example, one staff member recounted an instructor's concern regarding defaults:

> [wontfix] When you score assignments using the 4.0 scale, you are given the 4.0, 2.0, and 0.0 as prompts for entering in the desired percentages. However, since the client used percentages 60% and up starting from 1.0, and put in 0% for 0.0, all the percentages under 1.0 are drastically lower than he had intended—he recommends that we put in 1.0, 0.5, and so on forth to avoid this error.

One of the designers replied:

> [wontfix] We are deliberately leaving the interpolation open to the user's customization. There are so many ways that people do grading on campus, and there's no standard across the university or even across departments...

Online Grade Submission. The last change we discuss is the addition of an online grade submission feature to GradeBook, which would take the grade points in the final grade column and submit them to the registrar. As with the previous examples, the team believed they were working with fairly common user expectations; in this case, this was because the expectations were fairly well defined policies dictated by the university registrar:

> [dev] there were a lot of different policies and rules around what types of classes, or what types of grades specific students can get, the different types of classes that there are, I think there was just a lot of research we had to do to figure out what all of those rules were.

In addition to the relatively clear requirements, the developers also found that integrating the feature was straightforward:

> [dev] ...it was almost as simple as adding a link that would go to online grade submission, and then just making it aware of what classes were actually attached to that GradeBook.

Where the true challenge came was in coordinating with the "SDB," the legacy database containing student grades:

> [dev] the biggest road block in all this is that all the grades live in the student database. SDB. That's what they call it for short. And, we can't, we don't have access to those grades. Nobody has direct access really to the SDB... at the time we partnered with those folks who had access to SDB, and they created a series of web services...

Moreover, the team working on the legacy database was focused less on the user experience than desired:

> [pm] I mean I'll tell you that whole process was extremely difficult... They're Cobol based mainframe structures, which are really difficult. And in that process, the student team in creating the web service, really was thinking about representing the data in a sort of honest, accurate way, and not about the end user need, what the system needs in order to make the experience usable for the end user.

Once deployed, however, variations in grade submission practices emerged. For example, grade submission *delegates*, staff who could submit grades for multiple classes, faced significant delays in submitting grades:

> [pm] ... its completely a performance nightmare because there is no index, so it has to do a loop over the tables in the mainframe in order to figure out what classes you have grading delegate access to... Some people will do a click and in a few seconds, it'll come up with a couple classes, but some people... there just going to sit there and it might even just time out. And that was because there was no index to request a service change. Its just not going to happen.

Other feedback arose from the fact that the online grade submission was codifying registrar grading policies that had previously been less formal paper practices. For example, in one bug report, a designer recounted an instructor's need:

> [wontfix] She needed to submit a final grade for one student within 2 hours, because the student's financial aid was depending on it. However, she had 30 other students that she wasn't ready to submit... This puts her in a very sticky situation...

While it would have been theoretically possible for GradeBook to support such functionality, the team's only recourse was to surface the registrar's policies in the UI:

> [wontfix] The registrar does not let you do such a thing. That's why there's the X (No grade now). Unfortunately that is not much help to this instructor, but that's the way it is for now.

GradeBook's online grade submission feature was widely adopted, with 90% of all class grades submitted online in the last academic quarter of our observations. Unfortunately, GradeBook itself was seen by most staff as just an extra step to online submission and the registrar ultimately requested that GradeBook and online submission be separated. Unfortunately, most of the features users found useful for uploading grades (particularly Excel import), were too closely tied to the GradeBook data model to be reused.

6. DISCUSSION

The goal our case study was understand what constrains a software team's ability to address post-deployment user feedback in the form of the help requests and bug reports. We found that while feedback was a significant source of knowledge about user practices, translating this knowledge into changes to GradeBook's implementation was constrained by conflicts between heterogeneous uses of grade information in the user community and global assumptions made in the team's initial implementation. Ultimately, the information architecture inherent in the team's data schema was simply not expressive enough to support the diversity of information uses. Therefore, while GradeBook was adopted by many instructional staff, the team's interests in evolving the application to serve new adopters' new needs was hampered by the risk of breaking existing use cases, the costs of migrating existing data, and the unlikelihood of changing other infrastructure and processes over which the team had little control.

These results raise several questions about the role of user feedback in the post-deployment life of software applications. For example, would it have been possible for the team to somehow design the application in a more flexible way to serve a larger subset of the user community, without simply rewriting Excel? Is it possible, for example, that there was a degree of flexibility somewhere between Excel and the data schema the team designed initially that would have been expressive enough? If so, the question then becomes whether the team's oversight of this design possibility was a failure of the user research and requirements gathering. And yet, the team had already invested six months in its user research, far more than many commercial software projects; how much prototyping and iterative evaluation would have been enough?

It is also possible that the team could have done a better of identifying the assumptions they made in their initial user research and using the stream of post-deployment user feedback to test and validate these assumptions. Earlier detection of problems with these assumptions may have made it easier for the team to have addressed them, before the user community grew too large or the code grew too complex. For example, perhaps if LST had been more explicit about the limitations of their assumptions about the number of assignments and students faculty would add to a GradeBook, they could have designed testing procedures that may have revealed the performance problems earlier, before the performance limitations reached throughout the system's implementation.

Existing research on software design suggests several theoretical framings through which software processes might be devised to account for assumptions. For example, Naur described programming as building a theory of how a solution relates to the world [22]; design theory perspectives view software designs as *value judgements* [12], projecting "ideal" users [3] and expecting users to conform to them. For example, Friedman et al. suggest that systems ought to be free of bias by identifying it [12]. Similarly, Fischer et al. argue for escaping the user/developer dichotomy and empowering users to be their *own* designers [11]; but in doing so, teams may prioritize users who *want* to be empowered, but not those who want curated, pre-existing solutions. These perspectives may be helpful in designing new software processes that formalize and operationalize the identification of assumptions, helping software teams to think more carefully not only about the application's design, but also how assumptions are manifested in software architectures, testing plans, and triage processes.

While our case study primarily revealed evolution constraints related to modularity and heterogeneity, our results also suggest that the gathering of user feedback may *itself* constrain software evolution. Our findings show that by letting user feedback drive change, the GradeBook team mostly heard from *existing* users of the application, and even then, they mostly heard from a vocal minority which may not have been representative of the user community. Moreover, when these vocal minorities did provide feedback, change was usually denied, disincentivizing further feedback. This had the effect of *hardening* the original design, crystalizing it around existing uses, rather than future ones (in the same way users' workarounds can prevent software change [23]). Our study therefore highlights the importance of treating user feedback as less of a guide for what to change and more of a signal for the need for further research. In particular, user feedback should be a sign that users at or beyond the boundary of an application's idealized user [3] are struggling to adapt the software their needs.

Such signals should drive explicit studies of non-adopters of the system. These recommendations reinforce arguments that the point of data gathering is not to *drive* design, but inspire it [13]. Moreover, it also reinforces arguments that in some cases, the only way to better serve user needs is to abandon software [6], as we do with deteriorating physical systems.

One can also take a more positive view of the team's responses to user feedback: the team succeeded in anticipating many aspects of their user communities' homogeneous needs, implementing the application in a way that ensured additional grading scales and alternative workflows would either be easily added or supported by the Excel import. While the team faced tradeoffs between designing for flexibility and preserving simplicity, it may not have been possible to design an application that served everyone in their community; only serving some well, even if it means not serving others, may be an inevitable part of software design.

The implications of these observations in our case study on the larger concern of software evolution are many. For one, having a clear notion of who software is intended for is not only important in the design of software, but also in the architecting, testing, and evolution of software. Software processes should focus on ways of making the audience more explicit and finding ways of weaving the assumptions inherent in a design throughout an application's implementation and throughout a team's processes. Our study also suggests that an inherent part of triaging post-deployment feedback involves clarifying the values a team wants to uphold; without clarity, there is little to decide whether a potential change is important enough to risk the harm that changes might do through new defects and broken uses cases to the existing user community.

7. LIMITATIONS

As with any case study, our results should be generalized with caution. The team we studied did compete with other products, but for users, not for money. This could have affected how much weight was given to user concerns, relative to market concerns. The team was also focused on serving a university community to which it was directly affiliated with; this is in contrast to many other software development contexts, where software teams serve a client or a purchaser, rather than end-users directly. The team also followed an Agile process with two week sprints; the length of sprint might have influenced the size of changes that would be considered, relative to a team that works in 6-month cycles.

8. ACKNOWLEDGEMENTS

We thank the University of Washington's Learning and Scholarly Technologies team for their support and participation, as well as the University of Washington instructors and teaching assistants who agreed to be interviewed.

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF-0952733. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

1. Arnold, R.S. (1996). *Software change impact analysis*. IEEE Computer Society Press.
2. Bagozzi, R. P. (2007). The legacy of the technology acceptance model and a proposal for a paradigm shift. *J. of the Association for Info. Sys.*, 8(4): 244-254.
3. Bardzell, S. 2010. Feminist HCI: taking stock and outlining an agenda for design. *ACM Conf. on Human Factors in Computing Systems (CHI)*, 1301-1310.
4. Baron, J. (2000). *Thinking and deciding.* New York: Cambridge University Press.
5. Blackwell, A.F. (2002). First steps in programming: A rationale for attention investment models. *IEEE Symp. on Human-Centric Computing Lang. and Env.*, 2-10.
6. Buxton, B. (2007). *Sketching user experiences: getting the design right and the right design*. Morgan Kaufman.
7. Cartwright, M. and Shepperd, M. (2000). An empirical investigation of an object-oriented software system. *IEEE Trans. on Soft. Engineering,* 26(8): 786-796.
8. Corritore, C.L. and Wiedenbeck, S. (2001). An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *Int'l J. of Human-Computer Studies,* 54: 1-23.
9. Eaddy, M., Zimmermann, T., Sherwood, K.D, Garg, V., Murphy, G.C., Nagappan, N. and Aho, A.V. (2008). Do crosscutting concerns cause defects? *IEEE Trans. on Soft. Engr.*, 497-515.
10. Edwards, W. K., Newman, M. W., and Poole, E. S. (2010). The infrastructure problem in HCI. *ACM Conf. on Human Factors in Computing Systems*, 423-432.
11. Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., and Mehandjiev, N. (2004). Meta-design: a manifesto for end-user development. *Comm. of the ACM* 47(9): 33-37.
12. Friedman, B. and Nissenbaum, H. (1996). Bias in computer systems. *ACM Trans. on Information Systems* 14(3): 330-347.
13. Gaver, B., Dunne, T., and Pacenti, E. 1999. Design: Cultural probes. *interactions* 6(1): 21-29.
14. Grossman, T., Fitzmaurice, G., and Attar, R. (2009). A survey of software learnability: metrics, methodologies and guidelines. *ACM Conf. on Human Factors in Computing Systems*, 649-658.
15. Hollan, J., Hutchins, E., and Kirsh, D. (2000). Distributed cognition: toward a new foundation for human-computer interaction research. *ACM Trans. on Computer-Human Interactions*, 7(2): 174-196.
16. Janneck, M. (2010). Challenges of software recontextualization: lessons learned. *ACM Conf. on Human Factors in Computing Systems*, 4613-4628.
17. Kahneman, D.; Knetsch, J.L.; Thaler, R.H. (1991). Anomalies: the endowment effect, loss aversion, and status quo bias", *J. of Economic Perspectives*, 5(1): 193–206.
18. Ko, A. J. DeLine, R., Venolia, G. (2007). Information needs in collocated software development teams. *Int'l Conf. on Soft. Engr.*, 344-353.
19. Lehman, M.M. (1980). Programs, life cycles, and laws of software evolution. *Proc. IEEE* , 68(9): 1060-1076.
20. Lindgaard, G. and Chattratichart, J. (2007). Usability testing: what have we overlooked? *ACM Conf. on Human Factors in Computing Systems*, 1415-1424.
21. Nagappan N. and Ball B. (2005). Use of relative code churn measures to predict system defect density. *Int'l Conf. Soft. Engr.*, 284-292.
22. Naur, P. (1984). Programming as theory building. *Microprocessing and Microprogramming*, 15: 253-261.
23. Pollock, N. 2005. When is a work-around? Conflict and negotiation in computer systems development. *Science, Technology & Human Values* 30(4): 496-514.
24. Scott, K.M. (2009) Is usability obsolete? *ACM Interactions,* 16(3): 6-11.
25. Stevens, W., Myers, G., Constantine, L. (1974). Structured design. *IBM Systems Journal* 13(2): 115-139.