Paychex Labor Tracker

Team Openstache

Michael Nuzzo	Christopher Hossenlopp	Benjamin Nicholas	Mitchell Skiba

Paychex

Timothy Best and Matthew Difranco

Faculty Coach

Ryan Schneider

Project Overview

For their senior project, Team Openstache was tasked with creating a prototype of an application that would allow people to manage their work punches from their mobile devices. This project was to consist of three components: a cross-platform mobile application, a server, and a database. The team was also tasked with giving a presentation on the project to a number of Paychex managers at the end of the project.

Purpose

The purpose of this project is twofold: to develop a prototype for a product that can be released and to provide an example of a way to expand their mobile strategy. Paychex had a very limited mobile strategy at the beginning of this project. The project is designed to showcase how mobile platforms could be leveraged by Paychex. It does this by taking some sample use cases for Paychex' services and adding features possible on a mobile device, such as location tracking and schedule management. By developing this application, the team was helping prove what could be done in the mobile environment.

Scope

The scope of this project changed multiple times over the course of this project. As described in the project proposal document, the project was a complete product. The proposal version of the project described a mobile application, web administration, and database server. In this version, the mobile application would contain the ability to punch in and out with the location tracking, a list of existing punches, a schedule of jobs, and a route through all of the jobs. It was also initially thought that the project was to build a full and complete product set. As a result, the team removed the scheduling and map features from the scope and focused on developing the mobile application and administration web tools.

Shortly after development began on an application that could punch in and out and would have jobs associated with the punches (a two increment plan), it was made clearer that the project was a prototype that was being developed as a proof of concept mobile app. At this point the team agreed to shift focus with the administration tools being out of scope and the scheduling and routing features of the app being considered in scope. The idea of e-mail notifications also came up during meetings and has been integrated into the server.

Desired Results

There are a few desired results from this project. First, that Paychex accepts the prototype and continues its development. Second, that lessons are learned from doing this project will be applied both within Paychex and by the members of the team. Lastly, that the team members become well adjusted to the technologies used.

Basic Requirements

The requirements for this project changed as the project progressed. The requirements below are the requirements as completed by the team.

Functional Requirements

Mobile Application Requirements

1. Punch
2. Linking in punches to out punches
3. Job
4. Adding Jobs
5. Linking Jobs to Punches
6. Displaying Employee Schedule
7. Logging in
8. Logging out
9. Displaying route to job locations

Server Requirements

1. Database must have list of users
2. Database must maintain the list of punches
3. Database must maintain the list of jobs
4. The Server shall send notifications if a punch meets certain criteria
5. The database shall have the concept of companies

Constraints

Time Constraints

The team had to complete the development in the project over the course of the six month senior project. The team has four members who agreed to work several hours per week. This work agreement ran into problems that persisted throughout the project. The team also had to work on the project around their other coursework.

Technology Constraints

The sponsors put forth a couple of technology constraints at the beginning of the project. First, the team had to develop the mobile application in a cross-platform tool. Secondly, the server shall expose all functionality through JSON web services. The team also felt that a third technology constraint, that they should only use server technologies Paychex is currently using, should also be in place.

The first technology constraint resulted in the team deciding between Phonegap and Titanium. Since the sponsor expressed a preference for Phonegap, the team agreed to choose that option. Phonegap works by wrapping HTML and Javascript in a web container. The idea being that by taking this route would reduce development time.

The third technology constraint gave the team the choice between a Java web server or a Microsoft .NET server stack. The team chose to go with a Microsoft .NET stack due to its ease of learning, ease of set up, and the reputation of setting up a Java server being difficult.

As a result of the third technology constraint, the constraint that the server have JSON web services available was handled by the .NET stack.

Development Process

For the selection of the development process, the team had a few considerations. First, it seemed like the requirements were pretty well set. Second, a number of the team members expressed a dislike for a heavy plan or paperwork driven process. As a result, a non-agile incremental process was looked for. Eventually the team decided and the sponsor agreed to go with Goal-Driven Development.

Goal-Driven Development

Background

Goal-Driven Development (GDD) is a process which was proposed in 2006 at the 30th Annual NASA/IEEE Software Engineering Workshop. It is an incremental process which categorizes broad requirements as goals and has them broken down into their individual component tasks. These tasks are components that can be done by individuals who are specialized in working in a specific technology. The estimation is done once the tasks are determined. The method of estimation was not specified so the team decided to use a combination of Wideband Delphi and Planning Poker. The sponsors approved each planned schedule and the team agreed on a delivery date.[1]

Sponsor Notifications

The process did not directly address communicating with the sponsor beyond what the customer would need to approve. Each increment was discussed with the sponsors at the weekly team meetings. Once the priorities were settled on, the team moved forward with estimation. Once the team completed its estimation, the sponsor was notified about the estimates. If the estimates appeared to throw off the schedule too much, the team would discuss this with the sponsor at the next meeting to ascertain which goals were of a higher priority. The team would then release to the sponsors at the weekly meeting once an increment was done.

Team Roles

Goal-Driven Development designates the following roles: Programmers, Project Manager, Business Analyst, Software Architects, Requirements Engineers, and User Experiences Engineers. All student members of the senior project team are programmers. Michael Nuzzo held the role of Project Manager whose role is to assign the team's resources and manage the requirements with the business analyst. Mitch Skiba and Christopher Hossenlopp were the software architects with the responsibility for design on the whole project. Michael Nuzzo and Mitch Skiba held the roles of requirements engineers who worked with the business analyst to further develop the requirements. Ben Nicholas and Michael Nuzzo started out as user experience engineers, with Mitch Skiba becoming a user experience engineer and Michael Nuzzo leaving that part of development. Further, from the programming side, the team started out with Michael Nuzzo and Ben Nicholas working on the mobile application while Mitch Skiba and Chris Hossenlopp worked on the server. Towards the late middle of the project, Mitch moved to the mobile application corresponding to his change to user experience engineer. Shortly thereafter, Michael moved to the server temporarily.

Project Schedule: Planned and Actual

When the team started the project, their understanding of the priorities and broad goals of the project was not exceptional. The team thought that the goal of the project was to develop a

functional evolving prototype that had the support of the company. What came out around week four or five of fall quarter was that the project was a proof of concept of what could be done in the mobile space and that Paychex had not decided if they wanted to go forward with such a mobile project. This moved requirements that were more practical and less unique such as administrative tools from in scope to out of scope and moved requirements that were more advanced and previously moved to the end of the project as a stretch goal such as the schedule and routing into scope. The result was a drastic change in schedule. Once the schedule went through it's first major change, a few other smaller schedule changes occurred.

Original Schedule

The original schedule had the team working on mobile app features such as punching in and out and jobs, followed up by working on administrative tools for bosses and Paychex employees. The administrative tools were scheduled for the second quarter of work. Since the first quarter involved implementing punches and jobs, features absolutely necessary regardless of the priorities, it was decided to wait until the end of the quarter to change the schedule.

Schedule Drawbacks

There were a few drawbacks to this schedule. First, it did not account for our customer's actual goals. This was the ultimate reason for abandoning its schedule. It was also based on estimates made before the team's actual rate of work had been identified and only moved some of the requirements to stretch goals as opposed to completely out of scope for our phase of the project. Lastly, the initial estimates were not updated as the project went on, resulting in eventual schedule overruns.

Replanned Schedule

The replanned schedule has the same first quarter schedule. The second quarter schedule was changed to include a release for infrastructure tasks and general bug fixing, followed by a company manager features release, a release for the schedule and routing, and a release implementing the requested notification feature. General administrative views on the desktop were moved out of scope.

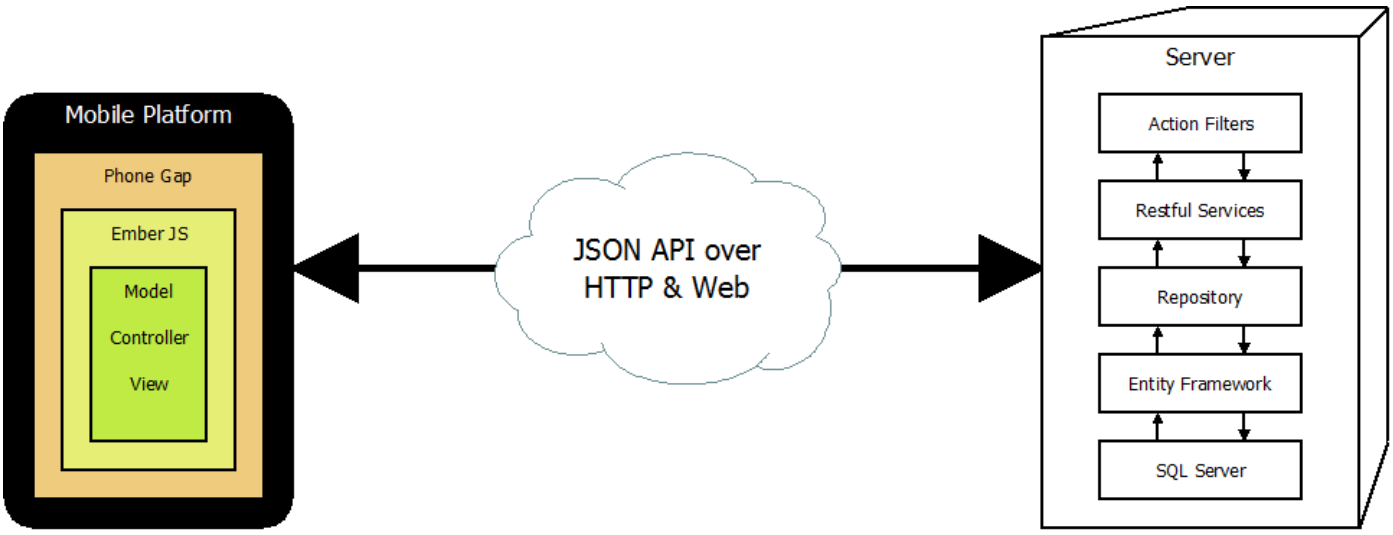
Ongoing Schedule Changes

The schedule has continued to go through changes. After the manager phone features were completed, the schedule and routing views were completed with a "dummy" back end that did not actually have any logic. Notifications were moved out of scope for this phase of the project but were moved back in since there was time for the server side to complete the functionality.

System Design

System Architecture

The system has two major parts to its architecture: the mobile application and the server. The two communicate over a restful JSON interface. Below is a diagram demonstrating this architecture.



Mobile Application Architecture

The mobile application was created using phone gap. PhoneGap is a library that wrapps HTML and JavaScript content in a native wrapper. Inside the PhoneGap application, our content is structured around Ember.js, a web client MVC framework.

PhoneGap was initially recommended as one of two viable cross platform options. It was chosen over Titanium as several sources recommended against Titanium. PhoneGap provides a lot of unused features. In addition to making a native wrapper, it allows the embedded JavaScript to uniformly access hardware features on the phone that are not part of the HTML5 specification, like the camera. In our case, needing only location, PhoneGap was much more than we needed.

The Ember.js portion of the client side is a page that the user stays on as they navigate around the app. Initially it was multiple HTML files, one for each view the user could see. This was changed for several reasons. Communication between the different portions of the application was hindered because when navigating to a new page, all in memory objects were discarded. This meant not only data, but navigation intent had to be stored in local storage. The initialization sequences for each view became very complicated, and reuse of views became extremely difficult. The other advantage to using a single page is a dramatic reduction in view switching delay. The app felt much more responsive after the switch to single page.

The costs of doing a single page PhoneGap application are a longer up front loading time and the requirement that all view templates be implemented in a single file. The initial load time of the application is currently unacceptably long. While they haven't been explored, dynamic script loading frameworks may help with this issue. The other cost, view templates being in one file, added a slight inconvenience to development. However, with more than two or three people working on the view at once, this cost may increase.

The scripts are organized by view, with global initialization and model definitions in separate files. Each area script represents of the views that can be navigated to by the user. It contains the controller that will react to user input and provide output to be displayed in the templates. It also

contains any specific view classes for interactions that could not be built off of the built-in Ember.js templates.

Server Architecture

The server is built on Microsoft's MVC3 and Entity Frameworks. MVC3 is responsible for routing requests and parsing parameters. Entity Framework is responsible for taking our database design and generating both a database schema and object-relational-mapper classes to interact with the database.

The MVC3 application uses filters to provide authentication and authorization. Provided through annotations, filters will be run before and after each request, and have the power to stop requests. A filter is used to restrict users from invoking actions they should not be allowed to, as well as parsing credential tokens and determining which user in the system is performing the action.

An early decision on the server side was to use a set of "TransferObjects" that would wrap database stored objects and serve as the basis for the restful interfaces. This allowed us a degree of freedom in separating the internal data representation from the representation that the client needed to use, and provided an easier way to have clients provide foreign key relationships. Every action takes a transfer object as a parameter instead of the database linked objects. The transfer objects allow access to the internal database object they represent. As a transfer object is manipulated it manipulates the internal representation accordingly. These manipulations range from simple pass-through to somewhat complicated internal foreign key management. A controller simply needs to validate the internal object, and save it into the database. We later learned some similar, limited functionality could be accomplished with annotations, but we still feel this was a flexible and safe approach.

To allow for testing, a repository layer was added that would abstract weather the database or an in-memory collection would be used to perform an action. Since most of Entity Framework is built around a tool called LINQ, the repository was easy to implement. Instead of needing to provide a gateway method for each kind of query, the repository can provide a single generic query-able object for each class that it can store.

Process and Product Metrics

Man Hours

Man hours is a metric required to be measured by the department. The purpose of measuring this metric is to ensure a fair amount of work is being put in by the team members and to make sure that the team is actually doing work.

The measurement of this metric evolved over time. At first we tracked the time in a spreadsheet. This led to inconsistent tracking as members would enter in hours for the same tasks with different descriptions and it did not present how the time was spent easily. Once the team's Redmine instance was running and the tasks were put into the tracking system, time tracking moved unevenly into Redmine. Once the team was accurately tracking time in Redmine, the time tracking process became second nature. Towards the end of the project, time stopped being

reported as team members became more focused on the fact they were graduating than on doing the project.

The results were that by the end of the project we could say that the team spent over 400 man hours on the project between implementation work, meetings, and other senior project work.

Average Man Hours Worked Per Week

This purpose of this metric is to measure the team's progress towards a goal. In this case, the team set a goal early on in the project to work 40 man hours per week, averaging 10 man hours per person. Early on in the project, we failed to meet the 40 man hours per week most weeks, but occasionally met the requirement. As the project progressed, this stabilized in the 20-30 man hours per week range, with little effort made to actually correct the shortcomings. The team instead focused on managing our estimated delivery dates as in some cases the team members could not actually meet the commitment to work 10 hours and the team members who could had completed their work.

Had the PM been more attentive, he would have recognized this as a warning sign regarding procrastination on the team and taken appropriate action, such as work meetings, to correct such problems.

Estimate Accuracy

The purpose of this metric was twofold. First, it would inform the team if estimates were very off and by how much the accuracy was off by. Secondly, it could be used to inform the team if a specific type of task was taking longer than estimated consistently.

In the first quarter, this metric pointed out that the team's estimates were becoming increasingly inaccurate. The team had estimated the project at the beginning and had not made many adjustments to those estimates. To combat the increasing estimate overruns, the team employed two strategies. First, the team switched from a single estimate at the beginning of the quarter and whenever the plan was changed, to an estimate at the beginning of an increment. In general the team found this to be more accurate. Secondly, the team started estimating the time to fix bugs and entering those estimates into Redmine. This ensured that all time was accounted for when comparing estimated hours to actual hours worked.

In general these changes worked. In the first quarter, one release took more than twice as long to complete as estimated in terms of man hours worked. In the second quarter, estimates were within 25% of actual hours worked with none of the overruns causing a delay.

Digressions per Meeting Hour

This metric measures how many times a meeting goes on a tangent and divides it by the length of the meeting. The purpose for measuring digressions is to track how off topic a meeting is getting and to draw attention to the fact that the meeting is getting off topic. The purpose of dividing the number of digressions by the meeting length is to prevent comparisons of the number of digressions for meetings that differ greatly in their length.

In the first quarter, this measurement was meaningful as it not only reduced the length of the digressions but provide a measure of how useful a meeting was being. It was found that the more frequently the team digressed (not measured), the less purpose there was to having the meeting. Over the course of the second quarter team meetings became increasingly short in an effort to make them more efficient and fit in time for the team to sit down and work together. A side effect was that actual meeting time was not interrupted by digressions any longer as the meetings tended to last at most 15 minutes.

Unit Test Coverage

Originally, the team had committed to using unit testing and this was to measure some of the effectiveness of the unit tests. As the team did not create many unit tests, this metric became rather useless to the team. If the issues that resulted in a lack of unit tests not occurred, the team would have used this metric to identify if there were unit tests that were useless and possibly parts of the system lacking sufficient tests.

Number of Defects

Over the course of the project, a total of 67 bugs were logged in the Redmine. The number of defects was used as an indicator of how the previous release had gone or of how the current one was going. For example, if a team member recognized a serious bug while working on the project before a release, that would indicate that there were problems with the current release. This would cause the PM to ask the people working on the affected areas if they thought this was going to cause a delay. If, on the other hand, a large number of defects were found by the sponsor at delivery, then that indicated that the release delivered was probably incomplete in some way. In one case, a number of defects were reported after the mobile application was rewritten. This displayed an incompleteness in the rewrite.

Defect Life Span

This metric was used to measure how long defects lasted. It was found that the average defect would not last through the next release cycle. This was good because it meant that the sponsors did not usually see a defect more than once. The only times this was not the case was for small features which were not thought of by the developers and which the sponsors did not mention much, if at all. One particular bug was the ability to punch out from the punch listing screen for punches without a matching punch.

Cyclomatic Complexity Analysis

For cyclomatic complexity analysis on the server, we used the built in Visual Studio metrics tool that comes with Visual Studio 2010 Premium and Ultimate editions. With regards to cyclomatic complexity, the code for the server rarely exhibited above a value of 10 and never concerned the team.

The team used the command line version of CCM to perform an analysis on the Javascript of the mobile application. While the analysis tool felt that there was very little risk in the complexity, the team did feel that this part of the project was in severe need of a rewrite. In this case, the analysis failed to identify issues with the maintainability. This is because the issues with the

application was repeated HTML from having each page be a separate file and the design of the Javascript making it difficult to add features to the code.

Product State at Time of Delivery

Completed Features

Mobile Application

The mobile application is mostly feature complete. Users can "punch in" and "punch out" with an assigned job. The user can set the date and time of a punch, with the default being the current date and time. The mobile application will get the GPS location and send that to the server with the time. If the device cannot get to the server, it will store the punch information locally. When the mobile application next makes a successful connection to the server, it will transmit any saved punches for the signed in user. All users can view a list of their completed punches, or see their punches on a map view. The user can view their job schedule, routes between jobs on a map, and the time to travel between jobs.

Users who are managers can create jobs and assign them to employees. Jobs contain information such as a point of contact, the address, and the start and end time of the job.Managers can view the punches of everybody in their company. They can also set the e-mail address to receive notifications at from the mobile device.

Server

The server side is partially feature complete. The database stores the information regarding jobs, punches, users, and all information relevant to those items. The database was not made multi-tenant, which was one of the earliest requirements. The server provides a JSON API which can be used to send data to and receive data from the server. The server also looks at punch data as it comes in and uses the rules that have been written to determine if a notification e-mail is to be sent to the employee's manager. The creation of new rules is easy to add, and the sending of e-mails is not tied to the notifications system exclusively, as required.

Incomplete Planned Features

Mobile Application

The mobile application is mostly feature complete in terms of features planned on. The only features that are not implemented on the mobile application in its current state are administrative functions that could be done from a phone. Such features include viewing the schedules for all employees and actively modifying those schedules. These features were moved back on the schedule to ensure the scheduling and routing features would be complete as those features act as a good selling point.

Server

The server is missing some features. The scheduling and routing is currently in a state where it only gives some very basic data for this feature and does not do any analysis or optimization. The server also lacks any web page to accommodate administration from a desktop web

application hosted on the server. This administration includes Paychex side client support, company side administrative tasks that are more complex, and other such features.

Added Features

A feature that was discussed but not added to the project is a kiosk application for tablet which would allow multiple people to punch in from a single location. This is a feature that Paychex could put on a roadmap if they decide to continue with the project.

Project Reflection

What Went Wrong

Phonegap

The project sponsor wanted development to be done on a cross-platform mobile library. Their first choice was Phonegap and their second choice was Titanium, which they had warned us away from. The team agreed to move forward with Phonegap at the sponsor's recommendation without evaluating how long developing native apps would actually take by comparison. While generally working decently well, there were a few major issues.

First, Phonegap changed name and package names, breaking the iPhone version of the project and requiring a reconstruction of the package. Since Phonegap works by wrapping an HTML5 web application in a local web view, scrolling was difficult to get working as there were few touch scrolling libraries and none that worked well for all platforms. This resulted in Mitch Skiba writing his own library for doing so.

A third major issue with Phonegap came with debugging. The team would do a bulk of the development on a Webkit based web browser like Google Chrome or Safari. This was done because these browsers used the same rendering engine as the mobile browsers and provided debugging tools. On a few occasions, however, the mobile application behaved differently enough to necessitate debugging on the device. Since mobile web browsers have no on device debugger, the team had to resort to Javascript notifications to track events in the application. This drastically slowed down the development of the application. Lastly, Phonegap is very slow. Even when it is not loading the Phonegap Javascript files it takes several seconds to load the application. Otherwise, the problems extend to the differences between Android and iOS devices that result in different sets of needs being addressed with some conflicts.

Team Time Management

The team had a number of issues with time management. First and foremost was procrastination. For most releases, there was the attitude that "this is easy" or "the estimate is only for the worst case" and no time was afforded for the case of an issue getting in the way of completing the task. This resulted in a few cases where the team had to delay release by a week or so. While this did not typically fully delay the project, it did increase tension amongst some team members. On one occasion this did result in a several week long delay (caused by other factors as well). The team agreed to some measures to cut back on the issue, but these measures were forgotten after a short period of time. Instead, the delays from procrastination were mitigated by extending how long the team estimated an increment would take.

The other problem was that every week at least one team member did less work than the others. This resulted in a stilted rate of development and delays occurred when team members were dependent on others to get things done.

Ember.js

Another problem that came up was with Ember.js, a framework we used for the mobile application. This framework was not particularly mature when the team agreed to use it and it created problems for team members who were less experienced with development. This caused delays in general, especially when a quirk of Ember became an issue. There were a number of cases where trying to work within Ember led to extensive delays and frustration.

Unit Testing

Unit testing is a very helpful thing. In this case, however, it was not very useful as almost no unit tests were written. This was the result of the ongoing procrastination issue at first. As the team became more and more rushed to finish each release, unit testing became a lower and lower priority. Eventually the team forgot to even think about unit tests all together. When the mobile application got heavily rewritten by one team member, the unit tests for it became useless and were not replaced.

What Went Right

Use of Wireframes

The team made a heavy use of wireframes throughout the development phases of the project. The wireframes were made using the tool Balsamiq. Ben Nicholas created all of the wireframes with Michael Nuzzo and the entire team doing a look over before sending them to the sponsor. The purpose of this process was to ensure that all planned features for the mobile application were thought of when doing the wireframe and would be included in the application's development. This process mostly worked, with one release failing to have wireframes that fully adhered to the goals of that release. This was the much troubled R2 release. Every release since has been on goal with the wireframes getting more scrutiny.

Redmine Project Tracking

Using Redmine to track project goals, time spent, meeting agendas, API specifications, etc. was a great help to the team. With the number of schedule changes that occurred, keeping track of the history through Redmine was easy as goals that were now out of date could be marked as closed and goals that were moved could be moved to a new release. The "Roadmap" view helped the sponsors keep an eye on what the future of the project looked like. The calendar helped provide a visual aid of how much time had been spent on the project. The "Issues" tab, when set up to show tasks by assignee and percentage worked, helped the team keep track of where each member was in their tasks. Team members would log time worked to tasks which Redmine would track and allow a report to be generated from, fulfilling our requirement to track hours worked.

Even though the project was short, in the end keeping the Redmine up to date was well worth it. The visibility it provided helped us communicate with our sponsors and in the end friction, if

any, was minimized. It also helped the project manager understand when the team needed to renegotiate the requirements.

Automated Build

The team used the Jenkins integrated build server to build the project after every commit and send out e-mails when the build did not finish compiling. Originally, Jenkins would run all of the unit tests as well and that would provide us with a better understanding of the status of the project, but with the lack of unit tests actually being written, it's main purpose became letting the team know if there was something wrong with the build in some other way. The primary cause of Jenkins build failures became files missing from commits or unnoticed syntax errors.

Backup Server on Student's Machine

Early on in the project, we had a number of outages on our department VMs. In one case, this negatively impacted the team's ability to push code to each other. As a result, the team set up a VM on Mitch Skiba's computer to act as a secondary repository if the first one went down and to back up the Redmine database. While this contingency was never used, it was good for peace of mind for the rest of the project after the team was unable to work for almost a day.

Adjusting Team Roles as Project Progressed

Over the course of the project, there were a number of changes to which members of the team were working on different parts of the project. Reasons for these changes included the current position being a misappropriation of skills, friction with the other person working on that part of the project, not enough work on one side with too much on the other, and problems keeping up with that side of the project. Each change had some initial slow down after the project got under way, but it did help keep the project moving on each side. In the case of friction between the two team members, changing the team roles helped prevent a potentially volatile situation from getting bad.

Incremental Process

Having an incremental process did help with the project. By letting the team focus on just the short term requirements, discussions about stuff we did not build did not occur after the first few weeks. It also helped our sponsor see progress, which we found to be important once we found out their previous senior project team had spent most of the project dealing with technology immaturity and had delivered very little. By cutting the project down into manageable chunks, we were also able to adapt when the team realized what the actual goals of the project were.

Would Do Differently

Team Work Time

One solution for the time management issues was to schedule time for the team to sit down and work. This was done on a few occasions and the benefits were recognized. By having everybody sit down in the same room and work, people readily communicate with each other about what they needed from each other.

Different Process

While Goal Driven Development worked, it did not quite work with the project. Since GDD is not specialized towards requirements or priority changes, it did not have a prescribed method for handling the changes that occurred. Given the changes that occurred and the lenience towards the due date that some project members had, an incremental and iterative agile process like Scrum would have been better. The change handling would have helped greatly and having an iterative process would have helped tighten the belt in terms of procrastination and identify the issue earlier. It would have also provided more metrics to gauge team members progress.

Native Applications

The team found that using Phonegap was an extra amount of unnecessary effort for a cross-platform application. The opinion has been expressed by some members of the team that going for a fully cross platform application is a frustrating a futile effort. Given our experiences with this project, the team has agreed that if we were to do the project again, we would instead write the mobile application natively for Android and iOS. It is felt that the rework done would be worth the ability to debug on device, performance improvements over Phonegap, ease of native development, and the ability to make UIs specifically for the devices

Evaluate Documentation More Thoroughly

The second biggest issue with developing the mobile application was the use of Ember.js. One of the biggest difficulties with working with Ember was the lack of sufficient documentation. If we were to do the project again, we would more thoroughly evaluate the documentation available for the technology instead of just the technology itself. This may have revealed some of the issues we had with Phonegap as well.

References

1. Schnabel, I., Pizka, M. 2006. Goal-Driven Software Development. SEW. 2006. IEEE Computer Society.