Tioga Tae Kwon Do
Student Management System

Team Kwondo
Curtis Cali, Nicholas Coriale, Andrew Deck, Andrew Vogler, Michael Washburn
Tioga Tae Kwon Do
Paul Mittan
Faculty Coach
Dr. J. Scott Hawker

Table Of Contents

Project Overview

Tioga Tae Kwon Do is a small Tae Kwon Do studio in Waverly, NY. The Tioga Tae Kwon Do studio has switched to digital records for membership and attendance within the last few years. In 2014 the studio worked with a senior project group to create a student management solution to track membership and attendance. The existing solution does not completely meet the needs of the sponsor, and a new, more complete solution is needed.

Our new solution focused on making performing daily tasks such as taking attendance and tracking student progression as easy as possible. The core functions of the system included registering new students, checking students into a class, ranking students up through different belt levels, and managing student information. The solution used a Windows desktop computer as the application's central location, which other interfaces (such as tablets) connected to. One key requirement was that the solution was not accessible from the World Wide Web, and is only available to the local intranet.

The tasks described above will be done by children ages 3 and up, all the way through older adults. The parts of the system that were student facing had to be designed in such a way that young children could use them without getting confused and needing the help of others. At the same time the system had to be easily usable for the adult students and parents that will be using the system on a day to day basis. Last of all, the non-student facing administrator and instructor functionality had to be designed in such a way that user actions could be completed efficiently, and easily, on both a tablet and desktop.

The available time to design and develop was roughly nine months, spanning from August 2016 to May 2017. To start, we spent two weeks on requirements analysis. After that, another two weeks were on software design. From there, we began two week iterations during which we would wireframe, design, and implement features, for a total of 10 iterations. At the end of each iteration, we delivered a working release to the sponsor. The first official release was delivered on April 13th, 2017. This left a window after development was finished to perform necessary maintenance tasks and critical bug fixes.

Basic Requirements

The Tioga Tae Kwon Do Student Management System focused on a small number of key requirements. The system needed to provide a streamlined way of registering new students for programs and recording all necessary student information. Once students are registered, the system provides a simplistic and time efficient way of checking students into a program. Attendance is tracked over time and these records are viewable by the administrator. The administrator is also able to edit student information, including the student's name, address, and emergency contact information. Additionally, the administrator is capable of creating programs, belts, and stripes to be used throughout the system. The system is capable of exporting all of its data, and importing its data from a previously created backup.

The common user of the system is anywhere between ages 3 and 50. The system was built with a focus on keeping tasks performed daily, such as student check-in and viewing student information, as simple and time efficient as possible. The data gathered from the system can be exported to a common format such as CSV, allowing for it to be imported by Constant Contact and Point-of-sale software.

After requirements elicitation the team worked on a domain model[1] to make sure the that the team had a good understanding of the domain and the requirements. This domain model was then discussed with the faculty coach and the sponsor.

Constraints

The primary constraint that affected the design, implementation and delivery of the system was the roughly 30 weeks (August to May) available to work on the project. Within those weeks each of the five team members had other academic responsibilities, meaning each member could only contribute 8 to 10 hours of time to the project per week on average. This constraint had the biggest impact at the start of the project when the team determined what was in and out of scope, and how the system would be designed.

The second biggest constraint the team faced was the operating environment. The system had to be designed to be hosted on a Windows 10 machine and accessed over a WLAN network exclusively, with the client interface being a Windows tablet. This constraint mostly impacted our technology choice, as it is not always easy to install software on a Windows based machine. However, the constraint that the most common interface was going to be a tablet also had a significant impact on the system, as designing mobile websites is complex. This constraint also required the system's user interface to account for multiple different screen resolutions.

Development Process

The development process started in the first week of being placed into our senior project team. In that first week, the team met with the project sponsor to review his needs and goals for the system. In doing so, the team learned that the sponsor had a solid understanding of what he wanted the system to do, and what he felt was out of scope for the system. The sponsor's knowledge of what should be in and out of scope can be credited to him working with the previous senior project team and learning from that process.

Given the well known requirements for the Tioga Tae Kwon Do Student Management System, the team chose the evolutionary delivery model. Evolutionary delivery is an iterative process in which time is spent upfront creating a software concept, then eliciting requirements, and then designing the system before diving into iterations of development. Once the team had decided we wanted to use evolutionary delivery, we presented it to the sponsor who approved it immediately, excited to be getting a working release of the system every two weeks.

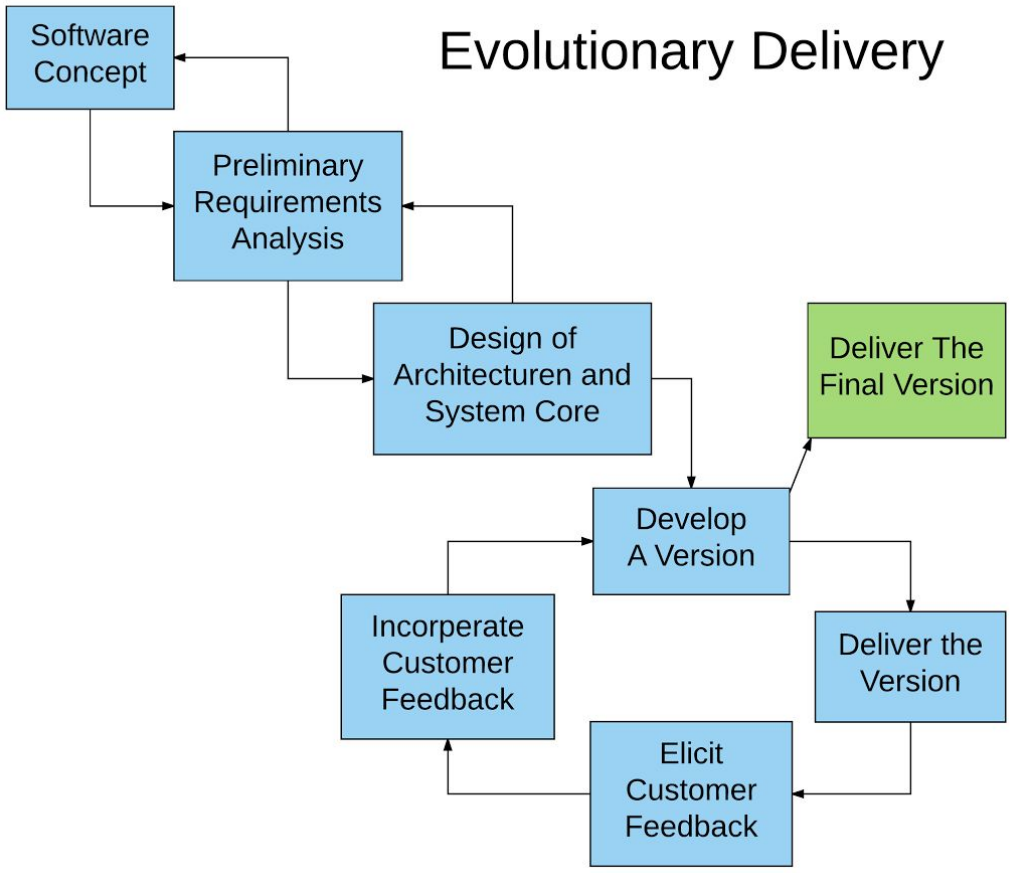

Figure 1 - Evolutionary Delivery

Evolutionary delivery was chosen not only because of upfront knowledge of requirements, but because the team wanted to elicit feedback from the sponsor on a regular basis. This frequent feedback from the sponsor played a key role in asserting that the system was satisfying the sponsor's requirements. The use of iterations in development allowed the team to produce a software prototype during each iteration. This prototype could then be shown to the sponsor who would give us his feedback as well as questions, comments and concerns. All of this information was then taken into account during the next iteration, where the team would address the sponsor's feedback from the end-of-iteration meeting.

Our team only saw the need for two roles, team coordinator and developer. A team coordinator was appointed to make sure that all deliverables were submitted, to lead both team and sponsor meetings and to make sure that everyone had work to do and was getting it done. The first week of senior project Nicholas Coriale was appointed the team coordinator, and thus everyone else was a developer, including Nick as a secondary role. A developer's role in the project included designing, implementing, documenting and testing the system. Developers also had to review and test other developers work and attend team and sponsor meetings every week.

Project Schedule: Planned and Actual

We planned our project to consist of four phases. First, we wanted to do one week of developing a software concept. That would be followed by a two week period of requirements analysis and gathering. After that, we planned to develop a solid design for the system in another two week period of time. The last phase we planned for was development. We planned on doing 10 two week iterations of development, at the end of which we would deliver an official release. After we developed requirements, we planned out the features we wanted to accomplish in the first five iterations, knowing that more requirements would be added as we completed iterations.

Planned Iteration Schedule

Iteration	Planned Tasks
Iteration 1	● Class Attendance ● Registration ● Deployment
Iteration 2	● Integrate and Polish UI/API for Class Attendance and Registration ● Deployments ● Class List ● Student List
Iteration 3	● Instructor Check-in ● Student Page ● Edit Student
Iteration 4	● Registration Pictures ● Student Attendance Tracking
Iteration 5	● Add Class

	• Import/Export

Figure 2 - Planned Iteration Schedule Through Iteration 5

As we made progress completing the project, we were able to add further detail to our iteration plans. The team also had times where we had to push back certain features to later iterations. We adjusted our schedule and planned future iterations as far out as we could each week. This resulted in the following final iteration schedule, as seen in Figure 3.

Final Iteration Schedule

Iteration	**Planned Tasks**
Iteration 1	• Class Attendance • Registration • Deployment
Iteration 2	• Integrate and Polish UI/API for Class Attendance and Registration • Add support for multiple emails • Deployments • Class List • Student List
Iteration 3	• Instructor Check-in • Add Class • Student Page • Add support for switching active class
Iteration 4	• Student Pictures • Individual Student Attendance Tracking
Iteration 5	• Edit Student • Import/Export • Authentication
Iteration 6	• Finish Authentication • Add Pagination on Student List • Finish Import/Export • Class Attendance • Add Karma testing • Auto Deployment • Finish Edit Student
Iteration 7	• Waiver view • Partial Registration • Add/Remove Belts and Stripes

	● Checkin instructors ● Add/Remove Student to class
Iteration 8	● Picture integration with Device camera ● Change picture orientation ● UI Design ● Firefox browser testing ● Taking pictures of waivers with device camera ● User management ● Instructor edit page
Iteration 9	● Live User Testing ● Focus on automated tests ● Tablet Responsiveness Testing ● Modifying/edit Programs ● View belt/stripe history on student detail page ● Implement/fix all back navigation ● Redesign 2
Iteration 10	● All bug fixes ● Finalize styling changes ● Deployment at TTKD

Figure 3 - Final Iteration Schedule

As we progressed through our iterations we gained a lot of insight into requirements that were missing, as well as additional features that were requested from the sponsor. Simple data management features, such as adding/editing programs were missed during planning, as well as more complex features, such as taking student pictures with a webcam. Ultimately, we were able to plan our iterations accordingly and complete all of the features listed in the schedule.

System Design

Our student management system was designed from the ground up to operate solely within the internal network at Tioga Tae Kwon Do (TTKD). The sponsor stated explicitly in the first meeting that he did not want this system to be accessible from outside of TTKD's network. The full system, operating solely with the internal network, consists of three major components: a Database, a REST API, and a UI. All three of these components run on a Windows desktop computer located in the office in TTKD. The system can then be accessed from any device with an internet browser (with a focus on design for tablets) connected to TTKD's network or from the desktop itself. The allocation of the system can be seen below in Figure 4, showing what the sponsor's Windows Machine is hosting and the devices connecting to it.

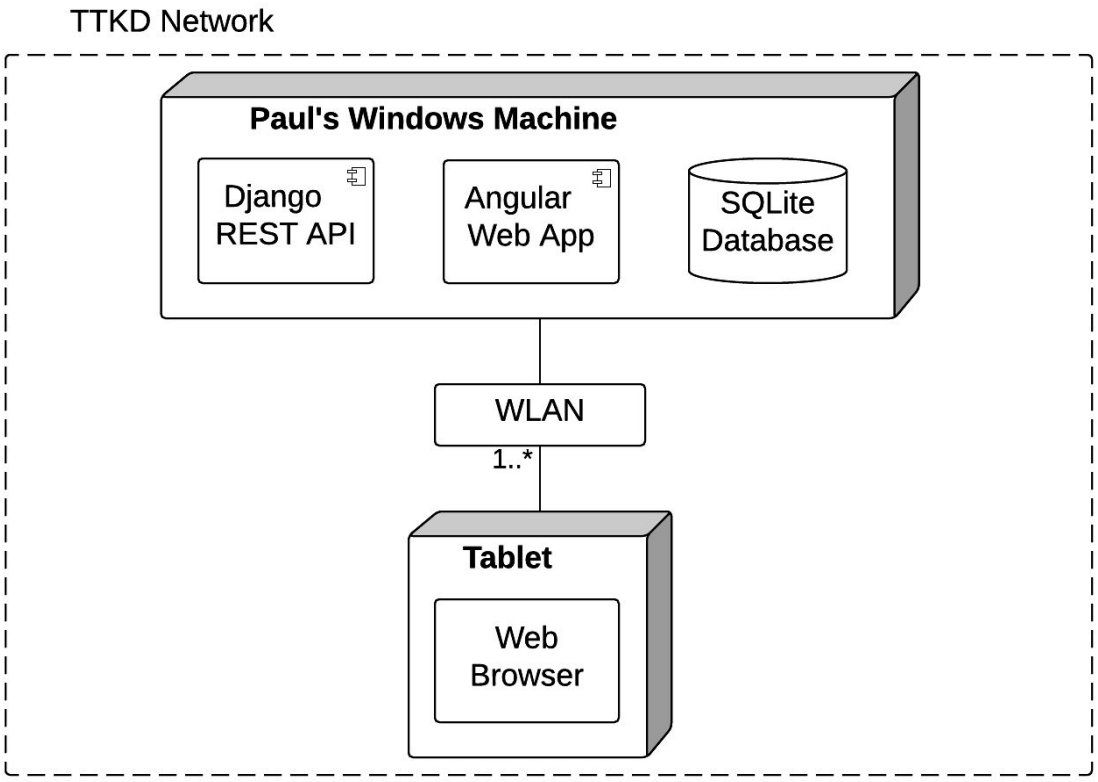

Figure 4 - Allocation Diagram

The team chose SQLite for our database technology because it offers the power of a relational database but also stores data in a file, making it much easier to deploy on the sponsor's machine. Our REST API was written in Python using the Django framework, and as Figure 5 shows the REST API acts as the connection between the UI that is displaying the data and the SQLite database that is storing the data. Django was our technology of choice for the web server because it could be packaged into an executable for deployment to any Windows machine. The team also considered Django a good choice because every member of the team was already familiar with it. This allowed the team to save valuable time by not having to learn a new technology, especially when we knew Django would fulfill our needs and requirements.

Django being an object relational mapper handled creating the database schema for us. However before the team did any coding we designed a database schema[2] to help understand all the data that the system will be storing and the relations between it. Creating this schema helped intial Django work progress easily. The design of the schema focused on storing a person's information and limiting the number of joins that were needed to get their information by keeping some foreign key references right in the person object. Overall the team feels that the schema is simple and well designed.

Finally, our UI was implemented using AngularJS and Bootstrap, which were chosen to create a responsive web application that could scale to various screen sizes. AngularJS in particular was chosen for it's flexibility, numerous add-on libraries, and the fact that Angular comes with a data binding feature. The application's main purpose is to collect and display data,

and as such the UI constantly needs to send and retrieve data from the REST API. Implementing this requirement without data binding would be both challenging and time consuming. AngularJS also had the added perk that three team members were familiar with it at the start of the project, reducing the time spent learning new technologies for the project.

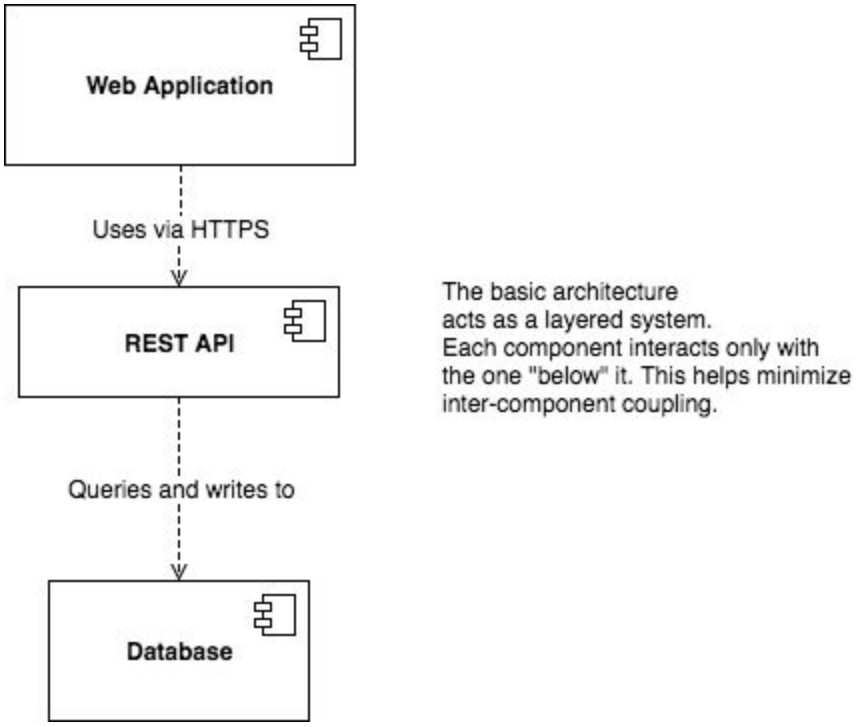

Figure 5 - Component Connector Diagram

Before the technologies outlined in Figure 4 were finalized, the team considered different alternatives that could be used for the database, the REST API, and the UI front end of the application. Because we knew that our web application would be primarily running on tablets, the team quickly decided that a single-page application (SPA) framework would be best to create a responsive and usable user interface. The primary SPA alternative that was considered alongside AngularJS was ReactJS. Although ReactJS has many useful concepts for creating responsive, usable, and maintainable user interfaces, no one on the team had any extensive experience using ReactJS in an actual software project. As a result of this circumstance, we decided that it would be better for the purpose of creating an effective and maintainable interface to use AngularJS, with which the team had more experience using.

To create the REST API, the team members focusing on backend work decided to use Python as the backend programming language. This was due to both the power and flexibility of Python, and the team member's knowledge of the language. After deciding to use Python, two possible frameworks were suggested. The first was Django, and the second was Flask, a Python framework for creating REST APIs. However, Flask did not have built in functionality for authentication and authorization, which was a requirement for our system. The lack of built in functionality of Flask caused the team to ultimately choose Django and Django REST Framework.

For the selection of our database technology, the team quickly came to conclusion that

the data our application would store was best stored in a relational manner. With this in mind, we rejected using the database technology used by the previous team, the NoSQL database MongoDB. Instead, the team chose to go with the SQLite, a relational database that is stored locally on a machine, instead of being run on a server.

Process and Product Metrics

Throughout the entire project our team tracked four metrics. These metrics were tracked, monitored and analyzed for analysis of how the project was progressing, and if an issue was discovered the team could make corrections. The first two metrics that were tracked were individual and team time/effort spent each week on the project. The other two metrics tracker were bugs per release and requirements defects per release.

Individual time tracking (per team member) was tracked so that the team could easily assert that every team member was putting in the minimum required eight hours every week, to ensure each team member was contributing equally. Each team member would estimate how much time they will spend on the next weeks work, and then record the actual time that it took to accomplish that work. Individually these estimates and actual times taken could be used to help oneself plan out what they could accomplish within a week.

Team time tracking (the sum of every team member's worked hours) was tracked so that the team could easily see which weeks were time light and which weeks were time heavy, as well as how the estimated team hours compared to the actual team hours. The Figure 6 below shows the estimated and actual hours for every week throughout the project. As you can see, the actual hours were almost always higher than the estimated hours, but overall the average difference between the actual and estimated values was about two hours. The first major spike at week 7 was due to the project's very first release to the sponsor, and because it was only the first two weeks of development for the project. The next spike can be seen at week 13, caused by the team having some major issues with our versioning software Git. Thankfully after a few extra hours these issues were resolved. Week 23 (Spring week 7) is high and off from the estimated for two reasons. The first is that all team members did a very poor job of estimating how much time their tasks would take. The second was that for week 8 the team had planned to visit the sponsor's studio, causing a push to get as much as possible done before the visit. Lastly, week 27 (Spring week 9) Mike and Nick put forth a large effort to complete more tasks than they had originally planned on completing for the week. They were only able to achieve so many hours because they were part time students. The difference between the estimated and actual hours can be seen in Figure 7 below.

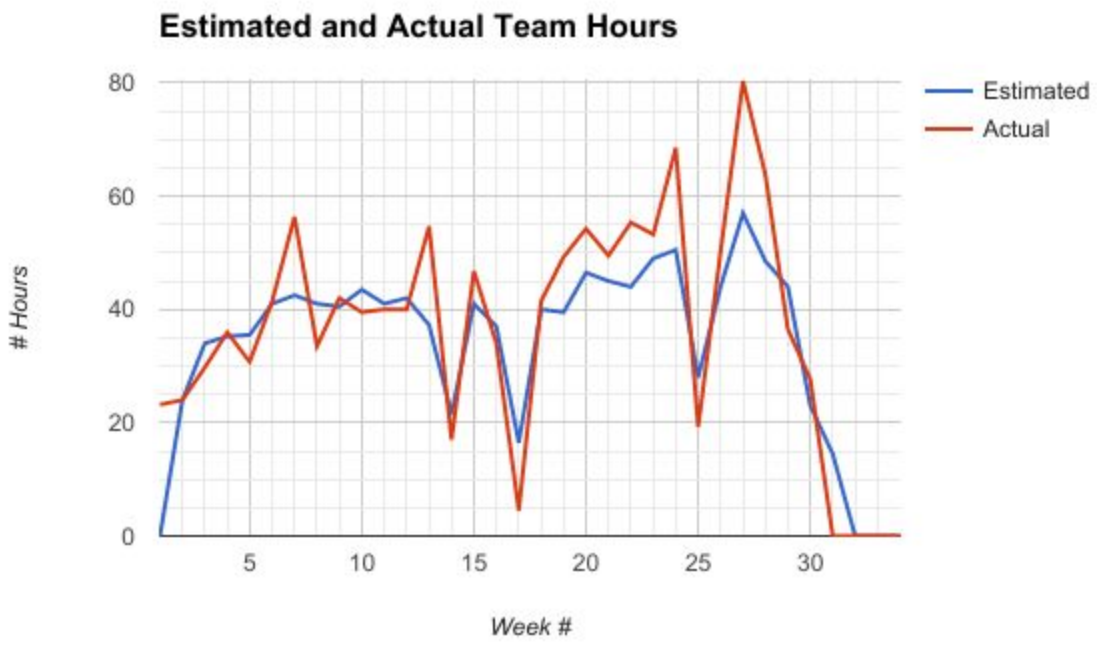

Figure 6 - Estimated and Actual Team Hours

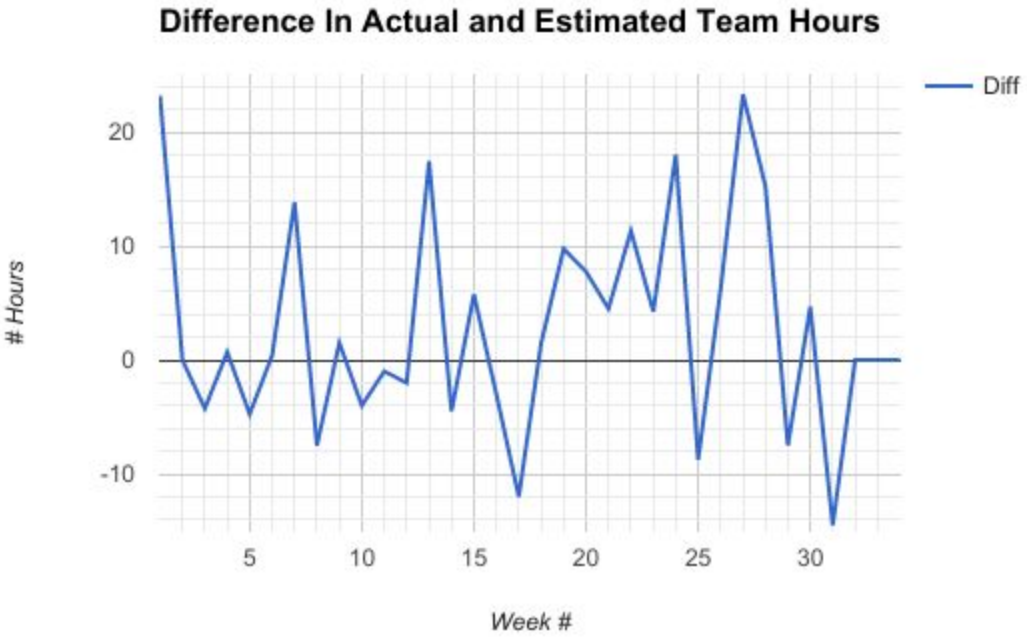

Figure 7 - Difference in Actual and Estimated Team Hours

	Total	Avg/Week
Estimated	1267.75	37.28676471

Actual	1354.47	39.83735294
Diff	86.72	2.550588235

Figure 8 - Estimated vs, Actual Team Hours Table

Choosing our third metric was easy. The team agreed that we wanted to track bugs per release. Tracking bugs per release allowed the team to see if our testing efforts were lacking for any release and to help us identify if we needed to improve our testing process or coverage. The team defined a bug as any code or functionality that does not work, or is broken. For example, if data that was intended to be displayed was not being shown, it would be counted towards the bug tracking metric. A bug indicates broken code, regardless of what the requirements might have specified around the issue. For this metric a bug is always recorded in the release that it was found, not the release that it was introduced.

The bugs per release can be seen below in Figure 8. Our goal as a team was to keep our bug count as low as possible, and overall the team was very happy with the total number of bugs that made it into releases. The team considers the bug count to be low numbers, and a sign of good testing done throughout the project by the developers both before and during code reviews. release 9 had the most bugs of any release at a total of nine, but the team still considers that bug count to be within reason. Iteration 9 included live user testing at the TTKD studio, which had the system thoroughly tested by end users. The live user testing was extremely useful to the project as we came away from it with a some requirements changes and discovered bugs that would have been harder to find in our development mindsets and environments. The nine bugs we found were also not all introduced in release 9, many of them had been around since previous releases and were never found until release 9.

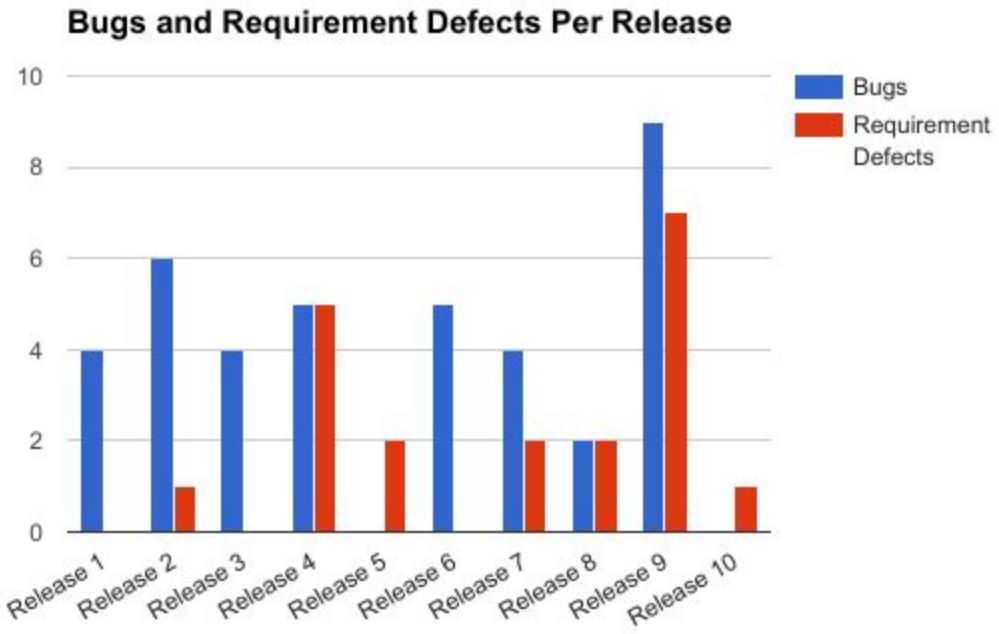

Figure 9 - Bugs and Requirement Defects Per Release

For the team's second metric we wanted something that we could use to see how well we are fulfilling the requirements that our sponsor had given us, while at the same time having a metric that could warn us of a missed requirement or scope creep. We defined a requirement defect as one of two things, it is either something we implemented which is not broken (it works) but is not what the sponsor wanted or it is a requirement that we failed to gather in our requirements elicitation, whether the sponsor had said it or not. An example of a requirement defect is if the sponsor had told the team to make a button red, but the team made it green. In this example the button is not broken code, however it is not what the sponsor wanted. If we had tracked all of these occurrences under bugs it would have been very hard to determine which bugs were caused by the team incorrectly implementing something or which bugs were written up because the sponsor changed his mind or had a new requirement.

The requirement defects started out very low across the first few releases as the system was simple and the team had a good understanding of what the requirements for the base of the system was. However as we got to release 4 and started to implement more complicated things, and things the sponsor might not have really known what he wanted we started to have a consistency of requirements defects. The team attributes most of these defects to the fact that more effort could have been put into the wireframes for features which would have helped us communicate with the sponsor before our misunderstanding or bad ideas made it into a release. That being said, the team was satisfied with the overall count of requirement defects and believes that we did a good job electing requirements at the start but could have done a better job and asked more questions when wireframing. Release 9 also had the most number of requirements defects for the same reason that it had the most number of bugs, they were found in live user testing. At the end of the live user testing the sponsor had seen the system working in the field and had a list of requirements that he wanted to change and these all were counted as requirement defects.

Product State at Time of Delivery

The first official release was delivered at the end of iteration 10 on April 13th. A second official release was given to the sponsor on 5/11/17 after a few minor bug fixes were added to the first official release. The last official release was delivered on 5/15 after another minor bug fix. With the handoff of the first official release the software was in its completed state, with the plan of only changing the product to fix critical bugs.

The software provided to our sponsor fulfills all of the core requirements that were agreed to at the start of the project and documented in the Requirements Document[3]. These core requirements include checking students into classes, tracking student attendance, and exporting student information. Not only does the system satisfy all core requirements, the system contains all planned features from the original requirements given by our sponsor, many of which were stretch features.

All of the delivered functionality has been tested thoroughly, both with automated end-to-end tests, and manual regression testing. The final state of the system includes a handful of known, minor bugs that were deemed to be very low priority, and thus were not fixed before

the development period was complete. The delivered product also contained documentation for deployment and installation, and a guide for running the program.

Throughout development, new requirements arose or were modified resulting in originally unplanned features. These unplanned completed features include adding belts and stripes to the system, multi-checkin mode on the instructor check in page and exporting the entire system database. One unplanned feature that was not implemented was integrating Google Calendars to automatically select a class to be checked in. This feature was proposed by the team, and agreed to by the sponsor. Ultimately the team decided that the effort required for this feature was not worth the development time required, and the feature was left unimplemented. Overall the unplanned features of the system were either proposed as usability improvements by the sponsor, or refinements of previously vague core requirements. These include different ways in which to view student data, and administrator management of users within the system.

Project Reflection

In retrospect, this project went well. As a team we were able to deliver a product to our sponsor that solved the problem he had. Overall one of the most positive influences to the success of the project was our process. The Evolutionary Delivery model allowed us to do a substantial amount of planning and design upfront, laying a solid foundation for development. More importantly the Evolutionary Delivery process allowed us to quickly get feedback from the sponsor on each release and played a significant role in the growth and final state of the software. Additionally, our choice of technologies was as helpful as we could have ever hoped as it did afford a way to satisfy all requirements without difficulty. The utilization of the Django web framework to create an API took advantage of its advanced and easy-to-use object-relational mapper. This saved the team a lot of time and effort when creating the API as compared to other technologies that could have been used. The use of AngularJS was another good technology choice that afforded a method to satisfy all requirements while at the same time being easy to learn for the two team members who had never used it before.

Another positive of the project was delivering a product to the sponsor that met all of the specified requirements, was on schedule, and met all of our usability goals. As a team we felt that our sponsor meetings were organized and well run. This had the benefit of keeping both the team and the sponsor in sync in regards to what was currently being worked on, and what was left to complete for the project. Our team was also

Conversely, there were a few aspects of the project that didn't go as well. We neglected to elicit all of the requirements of the system during our requirements analysis phase of the project. For example, we overlooked many data management forms that are crucial to the system, such as managing programs, belts, and stripes. These requirements were identified and satisfied eventually, but identifying them earlier on would have led to better planning and a less stressful development process for those features. Another thing during this project that could have gone better would be an earlier focus on testing. The team only manual tested during the first semester and relied on the sponsor to use the system for feedback. It wasn't until the second semester when the team put a bigger focus on manual testing, along with live user testing and automated tests. There's no question that the most valuable testing is right before the final release when all the features are implemented and intertwined, but the team definitely could have stood to gain from more testing earlier on, and that was something we noted after the first

semester.

References

1. Domain Model - Domain Model.jpg
2. Database Schema - TTKD Database Schema Final.pdf
3. Requirements Document - Requirements.pdf