LineUp

UPDATED 5/23/14
Requirements:
Process Description

Team Aphrodite
Nora Allison (efa91)
Bryan Djunaedi (dju90)
Stephen LaPlante (laplansk)
Evan Leon (ejl6)
Thomas Rothschilds (tgr4)
Simone Schaffer (simone09)
Nicholas Shahan (nshahan)
Evan Whitfield (evanw2)

CHANGELOG
5/16/14
Updated timeline to reflect project progress.
Added Bootstrap and SQL to programming languages/frameworks
5/5/14
Programming Toolset – design decision during SDS, updated to Javascript with AngularJS, Python with Flask because Scriptaculous is outdated.
Group Dynamics – updated to reflect architecture-related toon divisions following the SDS document
Timeline – updated to reflect completed milestones, ZFR accomplishments, and adjusted future milestones
Risks – updated to reflect SDS risks and Professor Ernst's comments on security risks

Software Toolset

Programming languages:

- Javascript with AngularJS
- Python Flask framework for route handling and database interaction.
- Traditional Web Programming (HTML, CSS, PHP

We have chosen to have our application use Javascript for the underlying dynamic content visible to the user, since this is dominant language of the web for client side scripting. AngularJS is an MVC framework (C is for collection, not controller here.) in Javascript that automatically updates the page when the underlying view changes. Python / Flask will be used as our server side framework because some of our team members have knowledge of Python and prefer it over PHP.

Version control:

- Git

We are choosing to use Git for our version control because more of our team is familiar with Git, and we will be using GitHub for some of our other requirements, so it will be helpful to have the version control tightly integrated with the website and other tools. Some of our team members also believe that Git is more widely used, and is a better tool to learn for that reason.

Bug tracking:

- Git Issue Tracker
- Google doc of known issues

Task Management:

- Google doc showing the weeks assignments.

Team members should update their own page after every team meeting, indicating what they completed in the last week, what they are working on, and what they will be working on the next week.

Group Dynamics

Our team is organized according to a semi-hierarchical communion model, with Bryan (project manager) coordinating communication between three toons: UI, front end, and back end. The UI toon will transition into testing after the website is mostly complete. The front end toon will handle model-controller interactions and is the stepping stone between the UI and back-end toons. The main task in beta for front end will be successfully implementing the Angular framework to display the queue states in-browser. The back-end toon will regulate route handling (requests from the user translated into Flask database-speak), the underlying queue object representation, and database management. Eventually, all toons will contribute to testing in some form or another.

Assignments are as follows:
Toon UI:
Evan Leon: UI implementation (webpage), fixing html/css linking issues, testing
Simone: UI design, testing, additional webpage implementation.
Toon Front End:
Stephen: front end implementation, testing.
Nicholas: front end implementation.
Bryan: Project Manager, UI design, front end implementation.
Toon Back End:
Thomas: database management, back-end implementation.
Evan Whitfield: back-end queue representation in Python.
Nora: back end implementation of Flask route handling.

Design decisions are to be resolved by majority. If no majority can be reached, then there will be a team discussion for at most 15 minutes, at which point the PM will have the deciding vote in the case of a continued tie. A majority vote is useful because it will ensure that each person is involved in the success of the project.

Disagreements about what a team member wants to work on can be resolved at the next team meeting. Each team member should recognize that there may be things they may not have wanted to work on, and should be willing to compromise.

Team members that are struggling with any task should ask for assistance from the PM. The most important thing with a blocked task is to get it “unblocked”, so that the team member can continue to be productive.

Schedule

This high level schedule outlines what will be completed by what days. It is up to individual groups to determine specifics. A very important milestone was the Zero Feature Release: By then, we expected basic queue functionality from front end to back end. This included being able to create, add to, and remove from a queue, with the queue state being stored on the back end. We were successful in this milestone.

Zero-feature release was the main focus up to this point. Things like dynamically keeping the queue up to date (and looking professional) on the front end and analytics on the back end were not of the highest priority. Now that basic views, models, functionality, and communication exist we are pursuing advanced features. Moving forward, our long term goal is now to define additions and produce a reasonable, functional, and tested system that meets most requirements over the next half-month (by the Beta-Release).

On April 18:
* UI team: prepared for Paper prototype. Had first draft of prototype. Additional team members helped as necessary.
* Front end: made final Javascript framework decision. Helped with paper prototype. Prepared for Architecture with initial views, models, and paths for state diagram.
* Back end: made final server framework decision. Helped with paper prototype. Prepared for Architecture with proposed database type and format

On April 21: Paper Prototype.
* UI team: Paper Prototype was completed.
* Front end: continued to work on Architecture.
* Back end: continued to work on Architecture.

On April 25: Architecture
* UI team: began HTML (testing and website creation). UI team and front end design started to work together, a defined outline emerged as to how the HTML and Javascript would interact.
* Front end design: “final” Architecture was completed.
* Back end: “Final” architecture. Database design, queries, and state diagram to handle front end paths was completed.

On April 28:
* Front end: Be able to traverse the website, with basic Javascript functionality. Be able to display queues, create/add to/remove from/delete them. Use a “fake” server, if necessary hard-coding data.

* Back end: Be able to serve the Zero-Feature front end interface, with database interaction. Recruit more team-members if necessary.

On May 2:
Zero-Feature Release.
Basic queue functionality was implemented. We could create queues (although not as simply as end-users will) and could access it, a tester could add/remove dummy "users" from the queue.

From here, both front end and back end teams have begun planning out what extra features to implement. Each team is working to develop their own timelines for implementation. The future schedule is up to each team to implement the additional features.

During this time use cases are being reviewed to confirm we are on the right track. We have continued to seek customer input to help determine features and basic layout.

Due May 9:
Beta Commit.
60% check in. This step ensures that progress is being made toward the upcoming beta-release. All components should be implemented at least on a basic level so that it can be completed on the beta-test day and reviewed.

The list of drafted portions at this point will include: The database will be set up enough to enable anonymous users, the route handler will be able to accommodate anonymous users and user self-removal, queue management will include queue creation, deletion, and settings, and the UI will be polish and have appropriate tests.

Due May 14:
Beta-Test.
80% check in. There will be a fully implemented version of each Beta Commit component, although testing may not be complete.

Due May 16:
Beta Release.
All major components in place and integrated at basic level. Each piece of code in the existing implementation (the beta-commit) will have been reviewed and tested by a different group member. The full documentation from the Beta Test phase will be complete.

Due May 23:
Feature Complete Release.

All major functionalities present, user can perform any tasks that are part of requirements; user documentation complete. NO FURTHER FEATURE ADDITION. Major bugs will have been located and eliminated. Both server-side and system-wide test suites will be implemented and at least partially completed. Implemented functionalities will include queue histories and analytics, searching, user settings, users can postpone their location in queue.

Due May 30:
Release Candidate.
Bug-free; documents in final form; test coverage verified for completeness. System will be verified to work across platforms. Security will be fully implemented on all components of the system. Test suites will by fully implemented and complete, including a UI test case suite.

Due June 4:
Final Release.
We will test the system on as large a scale as possible to see if traffic load affects performance and squash any remaining bugs, especially those belonging to edge cases. There will be Testing will be a combination of a coverage-review of the testing suites and a series of hallway tests and team stress-tests. All components of will also be verified and checked by quality control.

Due June 6:
Demo/Presentation.

Risk Analysis

1. Two Queue Member Types

Supporting two types of queue members -- those with LineUp accounts and those without -- may prove challenging. We will need different cases for the same requests, depending on if the member has an account or is a temporary user. This has a high likelihood of increasing complexity and causing bugs. During design, it was difficult to reason about how the system would handle a logged in member vs. one that is temporary, however, we did figure out a system for handling it. If problems arise (corrupted queues, corrupted user database), then we will eliminate temporary member features and require all members to have an account. When writing the SRS for this project, we did not realize the complexity that having two user levels would contribute to the design. Now that we have worked hard at sorting out the details of the design, we have a reasonable plan for wrestling those complexities. Nevertheless, we evaluate this to be a medium risk as we will need to continually make sure that all of our code handles both formats of user data. Should the problem become unmanageable during pre-release, the impact will be of medium level. We will focus on authenticated users and possibly drop the anonymous member from our build. Should it become troublesome post-release, we will add alerts forcing users to log in.

2. Choice of Framework

Choosing to use Python/Flask and Angular is a risk. We have only a recent understanding of these frameworks. There is a medium likelihood that much time will be spent simply learning and understanding the limits of the code. To mitigate this, we are trying to separate the frameworks from the coding. The Web framework and database details will be handled by specific modules in Python, while the queuing logic will be handled by others. As a result, the coding can be done in parallel, and no developer has to learn every part of the framework. This risk has been mitigated somewhat since the SRS, as some team members have already investigated the frameworks and have an understanding of how to implement our project with them, which has allowed us to get better estimates about the impact that this risk presents. We have experimented with a Flask python server prototype with satisfying results. It was easy to setup and understand. A prototype front end is in development using Angular.

3. Teamwork

We have 8 people on our team working on various different operating systems. Because of our large group size and intense deadline schedules, this is both a high likelihood and high impact risk. We may face problems with getting everyone set up with the same Python version and environment to do local machine tests. To mitigate, we may have to create a virtual machine image that everyone would install to do development and testing. We have subdivided our team into UI, client-side, and server-side groups, to modularize the work. This risk has changed since the SRS as we now have a better understanding of how our teams will be structured. To see if this risk is materializing, we will use status reports and member evaluations. We will encourage discussing team problems during meetings to solidify our project vision.

4. **Feature Creep**

A novel feature of the app is its wide range of potential users, leading to a risk of how to make it applicable to them without succumbing to feature creep. Currently, our risk is low, but without any mitigation, there is a high risk of this occurring. During our meetings, we have already observed that the complexity of the system has grown quite a bit since the initial concept was proposed, leading to different ideas about this risk since the SRS project. This additional complexity was anticipated and necessary. We may limit our customer range if feature creep occurs. We have stopped talking about new features for the time being and we are defining a concrete schedule which contains a date after which we will not add any more features to the product. To detect whether or not we are keeping this risk level low, we will compare the SRS and SDS documents with the current state of project and ask our clients to test the ease of use of our product weekly. Should the issue rise, however, we will let our PM selectively cut features with client feedback.

5. **Beta-Feature User Accounts**

We would like to include user accounts in our beta release. No one on the team is sure of how to have secure user accounts, however, making the likelihood of this being an issue high. If creating secure user accounts turns out to be more complex and time consuming than we expect, then we can implement basic accounts without security features during the beta release. Because the beta release will contain less-sensitive data, the impact of security errors will be low until we get to the later releases. To reduce risk, we have started brainstorming specific security issues, including protecting data when it is stored or transferred, validating requests, auditing data types that are passed to the client side (visible and not visible data), and sanitizing user inputs. We were not fully aware of this risk when we wrote the initial SRS due to focus on high level details. To detect if this risk materializes, we will try to hack our own server, as well as have validation for every user contact point (receiving data from or sending data to the user).

6. **Database Integration**

A risk here is that we could have trouble allowing our application to synchronize the information it is receiving for the position of people in the queues from multiple sources. Most of our team is not especially familiar with different database technologies, so we might have some trouble figuring out the right type of schema to use. In order to mitigate this risk, we could assign a team member to specifically focus on this portion of the application, so that at least one person could invest themselves in thoroughly learning the technology and feeling comfortable in this domain. In order to explore the seriousness of this risk, we will probably perform some prototyping of a simple usage of a database system with the type of data we will store.

Feedback from an external user will probably be most useful to us early on in the process, after we have completed initial prototypes, and have not committed too much infrastructure to a

particular design. We will hopefully solicit this feedback from the customer group we have in class, as well as talking to potential real world customers, such as CLUE tutors.

Our initial intention was to store the actual queue structures (the people in line and their order) in memory, rather than in the database, in order to avoid slowing down the application with frequent writes to the database. However, as Mike pointed out to our group, we were jumping to the conclusion that this would save time, and will not make a final decision until we have conducted timing tests and can back up whatever route we decide to go with hard facts.

7. **Security**

In addition to needing to safely store user names, IDs, contact info, and passwords, we must consider the possibility for abuse of the program in several cases: someone "spamming" a queue (repeatedly adding themselves), not being present when they are called or showing up late, or misidentifying themselves as the person at the front of the queue.

Mitigation: Communicate with customers (admin users) about the level of security they would expect in different contexts. Never store sensitive user information in our Github repository, or store it anywhere in unencrypted form.

Failure plan: In the worst case, there is no mechanism preventing someone from misusing the queue, just as there is no way of stopping a person from adding fake names to the whiteboard at a CLUE session or claiming to be whoever's name was just called to be seated at a busy restaurant. As far as protecting user information is concerned, failure is not an option and securing this sensitive data is an absolute necessity.