# LineUp

*UPDATED 6/4/14*
*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
6/4/14
Updated timeline.
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:
<u>Toon UI:</u>
Evan Leon: UI implementation (webpage), fixing html/css linking issues, testing
Simone: UI design, testing, additional webpage implementation.
<u>Toon Front End:</u>
Stephen: front end implementation, testing.
Nicholas: front end implementation.
Bryan: Project Manager, UI design, front end implementation.
<u>Toon Back End:</u>
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**

**April 7 – April 14**
**Bryan**: 5 hours setting up and conducting customer meetings, 3 hours coordinating and compiling SRS work from team members
**EvanL**: .500 days on UI (functionality)
**EvanW**: 0.5 days spent discussing architecture diagrams, 0.5 days spent researching/selecting frameworks to use for the application
**Nora**: Set up and configured github repository for project (1 day). Worked with Stephen on product description for SRS document (2 days)
**Simone**: 3 days spent creating initial LineUp mockups of all pages and possible interactions in Axure, 0.25 days spent meeting with extra customers to gather "real-world" requirements, 0.5 days spent making these fit the UI diagrams part of the assignment
**Stephen**: Stephen worked on the SRS over 2 days. He drafted a first copy and the following day Nora made changes and proofread it before calling it a final copy. **Stephen** spent about a day learning how to use both Git and Mercurial by reading through information pages on each one's website and completing an online git tutorial.
**Thomas**: Spent about 10 hours working on SRS. Evan W. and I came up with an initial timeline

**April 14 – April 21**
**Thomas**: Spent about 10 hours deciding on frameworks. Did a Flask tutorial and some javascript research.
**EvanL**: .125 days on UI (appearance). And .625 days on UI (functionality)
**EvanW**: 0.5 days spend assisting in creating a paper prototype of the product, 0.2 days researching flask
**Nora**: Installed and set up Python and Flask on my machine and learned Python/Flask (2 days). Served as a note-taker for the customer and application roles in the paper prototyping exercise (1 day)
**Bryan**: 10 hours making paper prototypes and their test routes
**Simone**: 0.5 days spent making paper prototype pieces, 0.25 days spent writing about our evaluation and helping with our reflection, 0.25 days spent meeting with our extra customers to gauge their liking of the UI, 0.25 days spent updating our UI design
**Stephen**: Rather than work on the creation of the paper prototype, Stephen spent a day reviewing the Java language, doing Angular tutorials, and setting up a server on Amazon's AWS.
**Thomas**: Spent about 10 hours deciding on frameworks. Did a Flask tutorial and some javascript research.

**April 21 – April 28**
**Bryan**: 3 hours documenting team structure and timeline on SDS, 3 hours coordinating and compiling SDS work on team members
**Thomas**: Spent about 15 hours on Architecture Design (database, server queue classes, interface).
**EvanL**: .250 days on UI (functionality), .375 days on system architecture, and .375 days on presentation preparation
**EvanW**: 0.5 days spent writing and editing portions of the SDS document, including creating slides for presentation

**Nick**: 1 days working with Stephen, Evan W, and Thomas to create architecture diagrams and class/object design, 0.5 developer days researching test frameworks for JavaScript code, 0.5 developer days editing SDS documents, 2 days building a prototype example page.
**Nora**: Learned Visual Paradigm and made object and sequence diagrams for SDS document (2 days), helped design back-end architecture and make design decisions about the server's behavior (2 days), contributed to test strategy and risk assessment for SDS (1 day)
**Simone**: 1 day spent updating the UI design, 1 day spent learning about Bootstrap and HTML, 0.25 days spent evaluating our risks, 0.5 days spent developing our frontend test plan ideals
**Stephen**:

**April 28 – May 5:**
**Bryan**: 2 hours setting up an AWS server and modifying the instructions for setting it up. 1 hour updating SRS team dynamics section
**EvanL**: .125 days presentation, .250 days web design (appearance), .250 days web design (content), and .375 days web design (functionality)
**EvanW**: 1.5 days implementing initial Queue classes in python
**Nick**: 0.5 days adding boilerplate client side code, organizing the github repository. 0.5 days updating the use cases from the SRS. 1 day researching automated testing and working on ways to email the group every night. 1 day working with Stephen and **Thomas** to get the production server to host our system and developing a procedure for deployment. 0.5 days editing documents.
**Nora**: Went thoroughly through Flask tutorials and started sketching out an implementation of the route handler (4 days). Incorporated Mike's suggestions in our SDS and SRS documents for re-submission (1 day).
**Simone**: 0.5 days spent creating our initial user- and developer-facing websites. 1 day spent creating our first HTML template pages. 0.25 days spent investigating different bug tracking tools. 0.25 days spent setting up the GitHub bug tracking tool for our project and filing the first bugs/tasks
**Stephen**: spent 1 hour reading about the merge functionality of git and 3 hours studying the already implemented angularjs code.
**Thomas**: Zero feature release: spent 15 hours on configurations. Wrote some very basic unit tests, helped configure the server to run a basic "Hello World!" Flask server.

**May 5 – May 12:**
**Bryan**: 3 hours learning AngularJS via online tutorial and trying it out
**EvanL**: .250 days web design (appearance), .250 days web design (content), and 1 day web design (functionality)
**EvanW**: 1 Day adding features to Queue classes, including a reverse index from QueueMember to Queue ID so that we could efficiently find the queues that a member was in.
**Nick**: 0.5 days building a controller and service for creating a new queue, 0.5 days building a controller and service for requesting and displaying popular queues, 0.5 days building a controller and service for creating a new account and logging in, 0.25 days building a controller and service for joining a queue, 1.5 days building a page a diagnostic test page for all these controllers to demo their use, 1 day connecting the controllers to the HTML UI, 0.5 days revising production and development deployment procedures, 0.5 days revising testing procedures, 0.5 days editing documents.

**Nora**: Integrated database operations into route handler and implemented several methods in debug_views and views (2 days). Tested and debugged Flask routing with url parameters (2 days). Incorporated Queue and QueueServer classes into route handler (2 days).
**Simone**: 1 day spent creating frontend test plans, 1 day spent user-testing our application, 0.5 days spent keeping the GitHub bug tracker updated and adding new tasks/bugs, 0.5 days spent preparing for the presentation
**Stephen**: Stephen spent 6 hours going over the production copy instructions and improving them with Simone. He also spent 3 hours configuring an auto-builder to replace the old cron job auto-tester situation which was decided to be unfit by the TA.
**Thomas**: Initial database utilities implemented. 10 hours. And Connected routes to database utilities: 10 hours

**May 12 – May 19:**
**Bryan**: 10 hours implementing administrator controller functions and some of the respective views in
**EvanL**: .250 days web design (bug-fixes), and .750 days web design (appearance)
**EvanW**: 1 Day adding support for analytics in the Queue classes, by making the queue store tuples of Queue Members and the time they were enqueued. This included updating the unit tests and implementing average and expected wait functions.
0.7 days writing unit tests for the methods in the queue classes
**Nick:** 0.25 days building a controller and service for Search/results, 0.5 days building a controller and service for requesting the queues you manage and own, 0.25 days updating controller for the creating a queue, 0.25 days building a controller and service for editing an existing queue. 0.25 days building a controller and service for requesting queue view details (queued and not queued). 0.25 days Setup Coveralls for code coverage, 0.5 days creating a header to be included on multiple pages.
**Nora**: Wrote addition implementations in debug_views and views (5 days). Closed github issues (2 days). Updated schema to provide info necessary for a display (1 day).
**Simone**: 1 day spent updating the UI design, including a color scheme re-vamp, 0.75 days spent user-testing our application and filing bugs/tasks, 0.25 days spent updating the GitHub bug tracker
**Stephen**: Stephen spent 6 hours going over the production copy instructions and improving them with Simone. He also spent 3 hours configuring an auto-builder to replace the old cron job auto-tester situation which was decided to be unfit by the TA.
**Thomas:** 20+ hours on implementations. Worked with Nick to solidify interface between client and server. Completed additional routes relating to login. Added login confirmations. Also about 8 hours figuring out why our Apache server wasn't properly serving our application. Basic functionality complete.
**May 19 – 26:**
**Bryan**: 20 hours implementing administrator controller functions
**EvanL**: .750 days web design (appearance), .375 days writing user test cases, and .375 days writing tutorials
**EvanW**: 0.2 days adding search functionality to Queue class, and
0.2 days adding and debugging search route
**Nick**: 0.5 days redesigning error handling and responses with Thomas, 1 day implementing the new error handling in create queue, edit queue, join queue, queue management, and create user

account pages, 1 day fixing bugs on all pages of the site, 0.25 responding to the code review and writing additional documentation for Angular code.
**Nora**: Updated project website for feature-complete release (1 day). Evaluated Note2Flash's developer instructions and website (1 day). Worked on uniform system for error handling (2 days)
**Simone**: 1 day spent updating the UI design, including a color scheme re-vamp. 0.75 days spent user-testing our application and filing bugs/tasks. 0.25 days spent updating the GitHub bug tracker.
**Stephen**: Spent 2 hours implementing with Angular the postpone feature of the product. He also spent part of an hour creating a new script that ran all of the tests on a local developers environment with a single line of code.
**Thomas**: 10 hours implementing Permissions and validation for Admin and Manager levels. And 10 hours debugging and preparing for Feature Complete release. Routes like postpone, leaveQueue, remove, managerPostpone, and managerAdd implemented. And 10 hours implemented and debugging error handling, and sending proper error objects back to the client from the server.

**May 26 – June 2**
**Bryan**: 20 hours finishing administrator controller functions, 2 hours implementing progress bar for queue info page, 30 minutes implementing integration with Google maps
**EvanL**: .125 days user testing, .500 days code review, .125 days user test cases, .250 days web design (content), .750 days web design (appearance), and .250 days web design (bug-fixes)
**EvanW**: 0.5 days working on making a plotter that wasn't able to make it to the final version. 0.5 days making additional unit tests for the Queue class and the QueueServer class to test additional features, 0.3 days working on organizing information about how time was spent on the project, 0.6 days working on improving plotter that will be included in a future release
**Nick:** 0.5 days to connect and debug QTracks analytics to be displayed in the queue view and admin pages
**Nora**: Wrote the back-end tests for the route handler (5 days). Learned pdb (1 day)
**Simone**: 0.5 days to add custom image functionality, 0.25 days to add queue status icons, 1 day to prepare for the final demo
**Stephen**: Stephen still needs to implement the feature in which queue information pages are updated automatically. He originally expected this to take 2 hours so he doubled it and now expects it to take 4 hours so he doubled it and now expects it to take 8 hours so he doubled it and no……..
**Thomas**: 20+ hours debugging queue settings updates, queue settings error messages, and queue administrator functionality. Finally fixed loading queue admins and managers into the queue server during application startup. And Temporary users finally implemented in full.

## 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.