Trillium Health Grant Management

Team Ulysses

Shannon Trudeau  
Akshay Karnawat

Brian To  
Matt Metcalf

Trillium Health

Russell James, AJ Blythe, Karen Elam

Faculty Coach

Daniel Krutz

Project Overview

Trillium Health provides many services to the communities of Rochester, including operating a full service clinic, pharmacy, needle exchange, social and care management services, community education, prevention, and many other things for a variety of high needs individuals in the Rochester/Bath/Geneva area. Since Trillium operates as a non-profit 501(3)(c) organization, they rely heavily on grant funding to continue their operations. Our software application works to manage the grant process (and all grant documentation) from start to finish. The system monitors, tracks, stores, and alerts people to complete necessary tasks as they are due, as well as help manage the initial process of searching for and applying to grants.

Trillium Grant Management System will standardize the grant attaining and maintaining process for Trillium Health. This is limited to notification of grant status, grant information, task creation, task reminders, and document upload, revision and approval. Grant status shall be communicated to the user by use of the dashboard by showing the status of the current tasks on a grant. Grant information shall be stored in the system and will be able to be viewed by clicking each individual grant. The system will allow for tasks to be created for grants and grant workflows. The system will notify users when a task is coming closer to its due date. The system will allow for upload of documents that are related to the grants. The system will keep track of previous revisions of documents so documents can be turned back to previous revisions. The system will allow for a process which requires grant owners to approve of task completion before they are finalized.
The main problem our product will address is the organization of their grant process. Before our product, each owner of a grant did whatever they felt was appropriate whilst attempting to manage the application, audit, and renewal process. There was very little sharing of knowledge, and it was easy for tasks to slip through the cracks. All documents were saved on local hard drives in folders with little organization or ability to communicate with the other employees of Trillium. The Trillium Grant Management system will centralize all the grants to one place so that anyone interested with the right permissions can see the current status of a grant and what the grant’s workers are currently doing. Visibility of the different grants processes as well as organization for each owner are huge issues that our product solves for Trillium.

### Basic Requirements

<table>
<thead>
<tr>
<th>#</th>
<th>Feature</th>
<th>Distinguishing Qualities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Grant Information</td>
<td>Users can see details about a grant’s funder, award amount, status, and workflows. This information is tailored to Trillium’s needs.</td>
</tr>
<tr>
<td>2</td>
<td>Tasks</td>
<td>Tasks can be recurring, which means that specific tasks will be re-included when an existing workflow restarts.</td>
</tr>
<tr>
<td>3</td>
<td>Task Reminders</td>
<td>Tasks can have reminders, which are emailed on a specific date before the task is due, customized by the user who created the Task. Reminders are also emailed out at the beginning of each week with the tasks that are due in the next week.</td>
</tr>
<tr>
<td>4</td>
<td>Documents</td>
<td>Documents are any form of digital media that Trillium needs saved as part of a grant or task. Documents may be financial (contains sensitive information) and can only be modified/viewed by those with the appropriate permission.</td>
</tr>
<tr>
<td>5</td>
<td>Document history and versioning system</td>
<td>Documents are also versioned, so older versions of the documents can be returned to if the changes were not wanted. Documents can be checked-out/in to prevent people from revision conflicts.</td>
</tr>
<tr>
<td>6</td>
<td>Active Directory</td>
<td>Users derive from Trillium’s AD; they do not have a separate login.</td>
</tr>
<tr>
<td>7</td>
<td>Microsoft Outlook Integration</td>
<td>Tasks appear on Microsoft Outlook calendars of the assigned users on the day which they are due.</td>
</tr>
<tr>
<td>8</td>
<td>Workflows and submissions</td>
<td>Workflows span the lifetime of a grant and encapsulate all tasks needed to accomplish an objective, such as an Audits or CFA.</td>
</tr>
</tbody>
</table>

### Non-Functional Requirements:

2
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Acceptance Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>The system shall ensure timely and accurate submission of information</td>
<td>The system should not take longer than 5 seconds to upload the document to the servers</td>
</tr>
<tr>
<td>The system shall standardize and keep a consistent the grant management process</td>
<td>Make sure that the process for each grant type is consistent.</td>
</tr>
</tbody>
</table>

The non-functional requirements speak to our commitment to the availability of our system and also the usability. Particularly, usability was a big concern for our sponsor. The user base at Trillium includes a majority of non-technically savvy individuals who are unfamiliar with new software and UI design. They wanted something that would seem comfortable to them, easy for them to learn and maintain a working knowledge of, and above all something that would be consistent throughout. This consistency continued when Trillium was assigned another senior project team for the second Spring semester. Trillium was impressed with what they had seen so far in our GUI design that they asked the other team to have a similar look and feel to our application. This way Trillium employees would not be obligated to learn a second completely new system.

**Constraints**

Because this is a brand-new system, we had free reign over the internal design. Implementation-wise, our application had to look up users from their Active Directory instance and must interface with Microsoft Outlook. Deployment-wise, the application must run on Windows Server 2008.

Since documents may be sensitive in nature and in such cases, must conform to HIPAA constraints, the document storage must be secured.

Must run on the latest versions of Firefox, Chrome, and IE. (We were unable to fully deliver this constraint, see below in Product State at Time of Delivery.)

**Development Process**

The process methodology we used was RUP [3]. We chose this methodology because it allowed us to complete a lot of up front scoping and requirements gathering the in the inception and elaboration phases. Another reason we chose RUP was because of the iterative nature of the construction phases. Iterative development [2] is something everyone on the team was very familiar with, however we felt the project needed to be completed in too short a time to use a strictly Agile methodology.
Our sponsor had no preference of development process. Their main concern was making sure that our process was transparent enough for Trillium to know the status of our project throughout the course of the year. Our communications with Trillium were based on weekly meetings. At first, our meetings were longer and more frequent because of the requirements inception phase. Furthermore, our process helped to promote integration, acceptance, and user tests at the end of each construction phase. We did this at the end of each phase as a way to highlight for our sponsor which features had been completed during that phase, as well as to obtain feedback from our sponsor regarding the look and feel of the system.

**Roles**

At the beginning of the project, we made a conscious decision to not assign rigid project roles to team members. The only roles we assigned at the beginning of the project were:

<table>
<thead>
<tr>
<th>Role</th>
<th>Responsibilities</th>
<th>Team Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sponsor Communicator</td>
<td>Primary handler of communication with the sponsor.</td>
<td>Shannon</td>
</tr>
<tr>
<td>Website Coordinator</td>
<td>in charge of all updates to the team SE website</td>
<td>Akshay</td>
</tr>
</tbody>
</table>

Throughout the progression of the first semester new roles emerged that became more official at the start of the second semester when we were starting to assign tasks once again. The definition of role within our team was not that the person was the sole worker on that set of tasks, but rather they were the “point-person” and responsible for making sure that work got done by the team as a group. Most of those roles are outlined below:

<table>
<thead>
<tr>
<th>Role</th>
<th>Responsibilities</th>
<th>Team Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team Coordinator</td>
<td>Wrote all agendas for meetings, kept track of all metrics and task distribution, maintained Long Term Planning Document.</td>
<td>Shannon</td>
</tr>
<tr>
<td>User Interface</td>
<td>Using Angularjs</td>
<td>Brian</td>
</tr>
<tr>
<td>Database</td>
<td>Creating the DB schema and implementing the structure within Hibernate</td>
<td>Matt</td>
</tr>
<tr>
<td>Outlook Integration</td>
<td>Tasks will show up on a user’s calendar in Outlook on the day which that assigned task is due.</td>
<td>Matt</td>
</tr>
<tr>
<td>LDAP/Active Directory</td>
<td>Retrieving a list of active users from Trillium’s AD and ensuring that each member of the organization could</td>
<td>Akshay</td>
</tr>
</tbody>
</table>
**Project Schedule: Planned and Actual**

Our project schedule was developed using the RUP methodology and our known allotted time frame. We used 5 week cycles for each phase of the project, repeating the construction/testing phases as necessary.

- Inception: Semester 1, Week 1-5
- Elaboration: Semester 1, Weeks 6-10
- Construction Phase 1: Semester 1, Weeks 11-16
- Construction Phase 2: Semester 2, Weeks 1-5
- Construction Phase 3: Semester 2, Weeks 6-10
- Transition: Semester 2, Weeks 11-16

The team made a strong effort to meet all the milestones originally put in place in the beginning of the first semester, however some milestones slipped through the cracks. Here is a list of the teams milestones throughout both semesters:

<table>
<thead>
<tr>
<th><strong>Milestone</strong></th>
<th><strong>Date Set to Complete</strong></th>
<th><strong>Status</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete Requirements Doc</td>
<td>End of Inception Phase/End of Week 5</td>
<td>We were able to complete version 1 of the requirements doc by this time, however there were a few draft versions that went back and forth between us and sponsor after the due date. This leaked into our Elaboration Phase</td>
</tr>
<tr>
<td></td>
<td>Semester 1</td>
<td></td>
</tr>
<tr>
<td>Complete Design Doc</td>
<td>End of Elaboration Phase/End of Week 10</td>
<td>This milestone was successfully met</td>
</tr>
<tr>
<td></td>
<td>Semester 1</td>
<td></td>
</tr>
<tr>
<td>User Testing Round 1</td>
<td>End of 1st Construction Phase/End of Week</td>
<td>This milestone was successfully met</td>
</tr>
<tr>
<td></td>
<td>1 Semester 2</td>
<td></td>
</tr>
<tr>
<td>User Testing Round 2</td>
<td>End of 2nd Construction Phase/End of Week</td>
<td>This milestone was successfully met</td>
</tr>
<tr>
<td></td>
<td>7 Semester 2</td>
<td></td>
</tr>
<tr>
<td>Code Freeze</td>
<td>End of 3rd Construction Phase/End of Week 11 Semester 2</td>
<td>This milestone was not successfully met. Tasks from C3 and remaining tasks from C2 took up a lot more of our time than anticipated, and we were coding until Week 14/15</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Final Deploy to Sponsor</td>
<td>End of Transition Phase/End of Week 15 Semester 2</td>
<td>This milestone was successfully met</td>
</tr>
</tbody>
</table>

For our first semester, the team was able to stay pretty closely on track. Aside from it taking a little longer than anticipated to nail down our final Requirements Document, we were able to stay pretty close to on track, and had a full functioning thin slice of our application up and running with our continuous integration server, Jenkins. This was really helpful when moving into the second semester, and allowed us to take a well deserved break over interim. The second semester was a little more rocky than we had hoped. Due to some loss of team motivation the weeks leading up to Spring Break, we got less completed during our second construction phase. These tasks bleeding over into our third construction phase really delayed our progress. We were unable to meet our Code Freeze milestone in week 11 and instead were coding and bug fixing up to the last minute.

**System Design**

For our application, we decided to write a web application to address their requirements. We used a traditional client-server model as a basis for our architecture. The client-side will run on the supported browsers while the server-side will reside in a Tomcat container hosted on a Windows Server 2008 instance.
The architecture consists of 3 main components, the client layer, the domain/application layer, and the data layer.

**The Client Layer** - The user will have an access to the application through a browser. The acceptable browsers are Chrome, Firefox, and Internet Explorer. This will be an AngularJS application that will send and receive information to and from REST interfaces in the application layer.

**The Application Layer** - consists of three major modules (Services, DAO, and Models). This layer is uses the Spring Framework to integrate everything together.

- **Services** - REST interfaces that the client will send information to. After processing the request, it will send information back to the user in a JSON format. This module has a Jersey REST package dependency that will be used.

- **Data-Access-Object (DAO)** - Data objects that the services will use to interact with the database. The DAO will perform the basic CRUD operation plus additional operations that are needed to query from the database. DAO module will be able to use LDAP and interact with that database to get the user account information and authenticate them using Active Directory.
Models - Components with fields that will be used for anything within the application layer. They will have the JPA and Hibernate annotations that will allow the application to persist data objects to the database.

The Database Layer - this is where all the grant information will be stored, along with the documents and any other relevant information pertaining to this application.

Technology Choice

When brainstorming technologies to build a web application, we started with this initial set:

- Ruby on Rails
- Node.js
- C#/ASP.NET
- Java/Spring

Ruby on Rails is quick and easy to scaffold an application together, set up database connections, and backed by a vibrant community and wealth of supporting libraries. However, Rails may be quite slow on a Windows environment. In addition, not everyone on the team is familiar with Ruby and Ruby on Rails, which may increase the ramp-up for members quite significantly. Furthermore, the Rails way of building applications does not lend itself to creating single-page applications.

Node.js is a very powerful, scalable, and trendy framework. It also might be helpful since the front-end and back-end languages are the same.

ASP.NET is also a very powerful, widely used framework. Since the development ecosystem is natively Windows, it lends itself very well to our environment.

Spring is also a widely used framework, has an extensive support community, and a plethora of libraries. On the other hand, the Spring ecosystem is prohibitively complex to newcomers not familiar with its conventions, flexibility areas, and philosophies when creating an application from scratch. Tools such as Spring Roo may have alleviated some of the pains, but no one looked into that alternative, so it was not used.

Spring was the only technology that did not displease everyone, we chose Spring as our web framework. Also, one spring is set up it tends to work and adding on additional spring features such as email is extremely easy.
Only one person had front-end experience, so the person with front-end experience decided to use AngularJS, because that was the most familiar. AngularJS is also widely-used, handles two-way binding, and many other niceties that make front-end development less painful.

**Build-Time View**

There are two build systems for this system: Maven and Grunt. The need for two different build systems is because neither build system adequately addresses all concerns, or cannot be accomplished eloquently and idiomatically.

Maven is the industry-standard build tool and dependency manager for the Java ecosystem. In this project, its responsibilities also include:
- Running local instances via `mvn tomcat7:run`
- Generating a deployable war file via `mvn package`
- Generating developer artifacts such as coverage reports, test results, and javadocs

Grunt, on the other hand, handles all development workflows dealing with front-end artifacts. Its responsibilities include:
- Scaffolding new pages
- Generating compiled css and javascript files
- Monitoring for changes in any less, javascript, or html file and recompiling upon detection

Unlike Maven, Grunt does not have dependency management baked-in. Instead, there are two dependency management mechanisms: one for run-time dependencies, and one for build-time dependencies. Npm is responsible for the build-time dependencies and Bower is responsible for the front-end dependencies. Npm modules are structured hierarchically with dependencies included in the module recursively. Npm tasks are straightforward to include in our Gruntfile.js build file. Bower deployables are specified as flat structures without dependencies. Bower modules are also strictly front-end dependencies. For this gap in responsibility, we are using two build systems. See [4].

For our templating engine, Thymeleaf, to find and amalgamate the HTML from our partial files, they need to be located in the WEB-INF directory, which is a Spring convention. In order to bridge the gap between these two conventions, both Maven and Grunt code is required to extract the partials and place them in WEB-INF. When developing front-end code with a server via `mvn tomcat7:run`, Grunt handles monitoring html files for changes and re-copying them into WEB-INF. However, when Maven builds the application, the Grunt artifacts are not generated within the webapp folder. Rather, they are copied to target-grunt and grunt is invoked there. With Maven’s notion of preparing resources, assets should be copied from target-grunt, not webapp, to
the war directory in target. This is because there is no guarantee that the asset compilation will have been done from within webapp, such as the case in our continuous integration server. The only time compiled files will be in the webapp folder is when someone invokes grunt localdev.

Front-End

Within src/main/webapp/resources/js, there are two folders: modules and shared. Modules store modules used within our front-end code. Those modules must not depend on other modules. Those modules comprise a fragment within the page, such as the left-hand grant navigation panel or the new/edit grant modal. The Javascript files in shared can depend on each other, but not cyclically. Modules can depend on any file in shared.

The shared folder was created for the need to reuse common code between modules. Ideally, modules can freely depend on each other. However, when developing without a server, modules can only depend on itself, not other modules. A possible workaround is to require all modules instead of just the module under development. The drawback is that it no longer isolates the code to just the module under development. The solution we used was to have a separate shared folder where reusable components or components that span multiple modules can reside in and be injected.

Each module contains 5 files and any number of submodules:
- controller.js
- controller-spec.js
- service.js
- service-spec.js
- partial.html

The spec files are tests for the respective Javascript file. The controller.js file is where the controller for this module is defined, via angular.controller(). The controller is responsible for handling inputs from buttons, inputs, and clicks, and updating data based on events that are broadcast. The service is responsible for any outstanding business logic such as how to filter tasks by user, due date, and so on, sending requests to the server, and receiving requests from the server. The partial is the HTML with Angular and Thymeleaf-specific syntax.

Since dependency cannot be cyclic, this poses a strong restriction on how controllers can communicate amongst each other. The approach we took is to broadcast an event from $rootScope and have the appropriate controllers listen for the broadcasted event. The advantage is that there can be multiple listeners for a specific message. The drawback is that
controllers can fail to perform a desired action because the message contract may differ between controllers. To a degree, this can also be viewed as tighter coupling.

The idiom for modals that create or update data is that there are two participating controllers: the panel controller, and the modal controller. The panel controller broadcasts a message that states “my add/edit button has been clicked”. The modal controller listens for that message and pops up a modal when detected. When an object is changed or created via REST call, the result is broadcasted as a message from the modal and the panel listens for that event. The panel then updates its stored model data accordingly.

Angular controllers when nested inherits its container’s scope. Initially, this sounded like a good idea for sharing data between related controllers such as the panel and the modal. However this also forces a dependency on the modal, requiring dependencies in the panel to be present in the modal as well, which makes mocking data for the modal much more difficult.

**Back-End**

We have three layers: controllers, services, and dao's. Controllers are responsible for receiving and responding to requests. Services are responsible for any business logic and DAOs are responsible for persisting/updating/retrieving model data to/from the database.

All services and dao's are split into an interface and an implementation. When code injects these dependencies, it should use the interface form. The separation is there because there may be a need to have different implementations depending on the runtime environment. Incidentally, this was the case. The development environment (on a personal computer) uses a different persistence setup than both staging and production. Likewise, the authentication required different implementations for all three environments.

All database actions: update, create, delete, and read, require a transaction to be present. This is handled in the service layer. Every method defined in the interface is annotated with @Transactional to create a transaction for that method. However, this setup implies two things: methods annotated this way should not call each other and the annotation should not be present in any of the DAOs.

Since all of the queries in the system are fairly simple, we decided to always use named queries. The advantage of named queries is that they are validated at application startup. If there is an error, it is detected there instead of at execution time. The drawback is that the queries cannot be dynamically created. However, this isn’t a problem for the queries we are currently using.
During development, we have effectively three different environments: development, staging, and production. The development environment is our personal computer. The staging environment is the virtual machine that is provided by the software engineering department. The production environment is the virtual machine on Trillium’s network.

The staging environment is Ubuntu 12.04 (Precise Pangolin). This environment in addition to hosting our staging instance also hosts our repository, continuous integration, bug tracker, and code review tool.

The production environment is Windows Server 2008.
We based our application off of a Spring archetype by kolorobot [5] which uses Tomcat 7.0 as the web server. We therefore decided to use Tomcat 7.0 as the container for our deployments, meaning that a Tomcat server will be running on both the staging and production virtual machines to host our application.

Since the choice of database to use is left to us, we chose a convenient database that is compatible with both environments. For this reason, we chose PostgreSQL.

Unfortunately, Active Directory service isn’t supported on Linux systems because it is a Windows-specific technology. So, we decided to use phpLdapAdmin as our authentication source since it seemed simple to use and fit the role AD has on Trillium’s environment. Locally, installing an LDAP instance is annoying since what users are stored on each person’s own computer can vary. Instead, we used a DAO implementation hardcoded for usage with local development, ie. hard coded users and passwords.

In case we need to clear out the application’s database or delete data, we decided to use a separate database to host Redmine and ReviewBoard data, mitigating the risk that anomalies in one database affect our development.

System Quality Attributes

To address our usability needs, we decided to try to consistently stick to a certain set of conventions to make the user interface more predictable and make the user feel in control. The decisions we made are listed below:

- All data manipulation such as creation and modification will be made through a modal. Non-modal modifications may skip this step if the change is not “dangerous” and can be done without additional data, eg. task completion and document checkout.
- Grant selection and data follow a master/detail pattern, where the master allows the user to select an aspect of a grant, such as information or tasks, and the detail is the specific detail for that aspect.
- Icons are favored over text for easier recognition and space saving. Icons also always have associated reminder text to remind the user of the meaning or intent of an icon.
- The following button color conventions are observed:

<table>
<thead>
<tr>
<th>Color</th>
<th>Blue</th>
<th>Teal</th>
<th>Green</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meaning</td>
<td>Primary action</td>
<td>Editing Modification</td>
<td>Creation</td>
<td>Deletion Removal</td>
</tr>
</tbody>
</table>
• Tabular or near-tabular representation of data are used whenever possible. This is because users are relatively familiar with Excel, and the tabular format will be familiar to them.

To address modifiability and future-proofing our project, we…
• …provided extensive on-boarding material which lessens the ramp-up required to develop on our application.
• …wrote front-end scaffolds to make page generation easier
• …used standard naming conventions to make navigation easier
• …wrote services and daos in a decoupled manner, where DAOs and services do not depend amongst themselves, and services strictly depend on DAOs.
• …provided infrastructure for both client and server-side testing

**Process and Product Metrics**

For our process metrics, we used time cards and tasks completed per phase. Time cards are how many hours we worked on anything senior-project related that week. This includes weekly meetings, presentations, and anything “mandatory”. Each task is captured as a Trello card. When the card is moved to our “done” lane, this indicates that the task is done. At the end of every phase, we take a look at the tasks in the done pile and count them to derive our metric.
For our product metrics, we used bugs / kloc, code line coverage, and the System Usability Scale [1].

Over the entire year, we spent over 1100 hours collectively on senior project. Each person spent between 200 and 400 hours in total, averaging at 280 hours per person, or about 7.5 hours per person every week. Our efforts collectively varied between 0 and over 60 hours per week.

If we examine the shape of the graph, we can see valleys at week 6, 16 to 20, 28, and 31. Week 6 and 28 are the two Career Fairs. The first career fair saw a larger dip in time than the second career fair. The valley from weeks 16 to 20 is the Winter break. We made a rule to not spend more than a few hours per person for the entire break since that was the last large break of our career. Thankfully, everyone followed through with the rule. The valley at week 28 was our Spring break, where again, everyone enjoyed themselves and didn’t completely dedicate their time to senior project.
The graph also has a few distinct peaks or crests at week 3, 15, 27, and 35. Week 3 is when the team started setting up environments, build servers, and other development tools needed to complete this project. Furthermore, this is when we started prototyping user interfaces. That perfect storm led to the crest at week 3. Week 15 is the last week of the semester and the end of our first construction phase. Since we were slightly behind, we had to use the budgeted “risk week” to finish any outstanding tasks, which led to the spike in effort. Also, this time was spent working on the presentation and practicing for the presentation. Likewise, week 27 is the end of our second construction phase, which caused the second spike. This time, we could see that some tasks would take much longer than anticipated, so that is why the week following also had a high number of hours. Weeks 34 and 35 were supposed to be the start of our testing phase. However, not all of our work was completed in our construction phases, so that led to the large effort in Week 34 and 35. After that, the work remained unrelenting until the bitter end.

After the first semester we were able to determine that each of our trello tasks took approximately 4 hours to complete based on our metrics. This helped a lot for the second semester because it helped us with planning how much time we had left and how many tasks we were going to be able to complete. We were able to identify scheduling problems at the start of the second semester instead of at the end of the project when it would have been a lot more difficult to negotiate scope with the sponsor.

Approximately 30% of our effort was spent in implementation, which is reasonable and resembles industry’s breakdown. Interestingly, both research and project management tasks took just under 20% of our effort, each. The bulk of our research efforts were in the inception phase and gradually decreased until our transition phase. Project management effort varied without a pattern. Research took a significant portion of our efforts because even though we all had some experience with our chosen technologies, no one understood it to the level necessary to create an application from scratch and debug problems.

Once we understood our problem domain, 10% of our time was spent creating and refining our design, knowing that mistakes at this point are the easiest to fix compared to discovering design problems downstream.

While we have a bug tracker (Redmine) in place and logged a few bugs, we did not log all bugs that we encountered nor aggressively looked for bugs, so the value in our bugs per kloc metric would not be useful. Right now, we have 2 logged bugs logged and at least one issue not logged. We have 3700 statements of Java code excluding test code, 3400 lines of javascript code excluding test code, and almost 3400 lines of HTML. All together, that is over 11,500 lines of source, excluding test source, which yields around 0.2 bugs per KLOC. Along with this code reviews greatly helped decrease the number of potential bugs in the system.
The following code metrics only apply to our Java code since the JavaScript doppelganger does not exist in our setup. The greatest number of statements in a single class is 148. The method with the greatest number of statements is 28. The greatest cyclomatic complexity in a single method is 9. The average cyclomatic complexity is 1.3.

User testing at the end of each RUP construction phase allowed us to get an accurate portrayal of the actual end-users feelings toward our system look and feel. The users would complete a test script which highlighted the main features which had been completed during the previous construction phase. After completing the test scripts, the users would complete the 10 question SUS metric survey and score each statement on a scale of 1-5. After processing the scores and averaging them over the 3 construction phases, we were able to obtain a SUS metric score of 85 (which is above an average SUS metric score of 68). Our SUS metrics from user testing indicated that our time spent on prototyping and interface refinement in the beginning of the semester was time well-spent.

**Product State at Time of Delivery**

The state of the product at time of delivery contains a strong, usable foundation. All major functionality is included along with the appropriate end-user and developer onboarding documentation to ensure this project is not thrown to the wayside.

We originally planned to include automatic integration with Outlook calendar. This became problematic for us. Instead, we have enabled users to get a link to a webCal that they can subscribe to, which will automatically add new tasks to a user's calendar. This requires a little more work for the user, but a lot less work for us.

Some other features we had hoped to accomplish but were unable to include: automatic workflows, task templates, and full system testing on internet explorer. While it was important for us to try and deliver as many features as we could, we felt our time was better spent ensuring the functionality of more essential features than having many buggy features.

One of the major defects found during user testing was the fact that some GUI functionality was not present when using Internet Explorer. This is problematic at Trillium because most users are comfortable with IE and have never used another browser. Furthermore, most Trillium employees are unaware how to install a new browser. It was decided by the team to not spend exuberant amounts of time on this known issue, as functionality on IE worked fine, and it would be possible for Trillium employees to switch their browser to google chrome in the future.
Project Reflection

There were a lot of successes for our team throughout this project. Here is a brief list of the things we feel went exceptionally well throughout our senior project year.

- **Keeping track of tasks.** For this we used mainly Trello, and our project manager was the Trello point person. She constantly updated the trello to make sure it was accurate, drew metrics from our trello boards, and addressed trello cards during every meeting. This helped with transparency so that everyone on the team knew what everyone else was working on.

- **Our metrics were on point.** We paid particularly close attention to our metrics to make sure they were accurate. This collection paid off in two ways. First, our task and activity metrics revealed to us that each of our trello tasks seemed to take about 4 hours to complete on average. This was useful for planning in the second semester. Secondly, collecting SUS metric scores at the end of every construction phase allowed us to have a really sleek and usable interface at the end of the project.

- **Code reviews were helpful** (we used review board). We found a lot of informal reviews better than occasional formal reviews. This helped up our code quality and also expose each member of the team to parts of the code they would normally not have interacted with.

- **Email updates went well** for the second semester. We weren’t able to meet as often in the second semester as we were in the first, so having functional email updates each morning was a good way to keep in touch with the whole team and our progress each week.

- **Continuous integration** was a blessing. Brian set up Jenkins at the very beginning of the project, and this cut a lot of time off trying to integrate all of our code together at the end.

- **TECHNOLOGY SPIKE!** Choose the right technology early, set up a stack, do it early. Brian did this and it paid off for the whole team exponentially.

- **Build in risk weeks.** We set aside three weeks for risk and we used all three during construction. This worked out well, but in hindsight we should have been more productive during these weeks and not considered them as much “bonus” weeks.

There were also some things that we feel did not go so well for our team throughout this project. Here is a brief list of the things we feel went exceptionally awful throughout our senior project year.

- Estimating time it will take to complete tasks. This was mainly a concern for the first semester. We tended to underestimate our task lengths.
• Overestimating the amount of time we will actually work on this project. This was mainly a problem for the second semester. There were a few weeks that we just burned out. This caused our project to have some delays.
• Not revisiting our original estimates. After discovering our first two problems listed above, we didn’t really do anything to try and combat that. Instead we just decided to work more on the project.
• Completely ignoring risk tracking after risk document was originally made. We built in those first three risk weeks, but after that we didn’t pay attention to which risks happened and which didn’t. We never brainstormed new risks or eliminated old ones that were no longer going to happen.
• Time is the biggest issue. Senior year is a struggle and with all the other classes we had going on and looking for full time jobs, it was hard to keep up with everything.

When everything was said and done, here are some things we wish we had done differently this year.

• Integration tasks or coding tasks being done together or in a group setting. This would have given everyone on the team a deeper understanding of the code base.
• Environment setup was a pain, a lot of tools, went ok but could have been better/ more efficient. If we had built in more time to go over this as a team, it would not have had to have been such a big time sink for Brian and his research spike.
• Don’t give yourself generous deadlines. Instead, make the deadlines early and go over a day. Work will fill the time allotted.

References