Stuff Plotter Overview
Stuff Plotter is a web application designed for a user and his small social circle to plan and coordinate events, parties, get-togethers, or even schedule personal reminders for oneself. Stuff Plotter uses your Google Account to access your basic profile information and sync your Google Calendar. You can schedule events on Stuff Plotter and it will be reflected on your Google Calendar, as well as see conflicting times for events already scheduled from your Calendar. You can even perform a simple word search of the location of the event due to having Google Maps integrated with Stuff Plotter.

Stuff Plotter recognizes that scheduling, planning and coordinating events can be a chore. So to differentiate itself from other scheduling applications, Stuff Plotter has implemented a leveling system and an achievement system. By performing certain tasks you can unlock achievements and show off your achievement badge to your friends. You can also gain experience and level up your Google Account by simply hosting or attending events, making friends or unlocking achievements. Stuff Plotter also has a real time news feed to display all your friends' progress. You can even view their unlocked achievements and race to see who can unlock them all first!

It is our aim with Stuff Plotter, to provide a social media hub where friends can host and participate events while having fun in hopes to encourage people to hang out with each other and promote a more social lifestyle.

Design Review PPT - https://docs.google.com/file/d/0Bx4B6pwWnX1uaGo4eGN4UWZXdGs/edit

ALL 19 USER STORIES [BASED UPON STATUS ON GITHUB]

https://github.com/UBC-CS410/orz/issues/milestones

01 - Richard Create Events (done)
02 - Matt- Invite Us ers (done/partially, scrapped the ability to cancel an invitation)
03 - Co-Host (Never implemented)
04 - Allen/Richard - Respond to Availability (done)
05 - Richard - Navigate Calendar (partially implemented)
06 - Allen - Link to more Information (done) -
07 - Allen - Communicate with other Users (done/partially, scrap the ability to send out private msgs)
-
08 - Allen - Like Button/Ratings (done)
09 - Search Events (Never Implemented)
10 - Richard/Allen - Set Schedule Time (done)
11 - Allen - Select Scheduled Time (Done/partially - no histogram, or suggesting time)
12 - Matt - User Profile (done)
13 - Moderators (Never Implemented)
14 - Matt - Befriend (done)
15 - Matt - Achievement (done)
16 - Matt - Level Up (done)
17 - Prizes (never implemented)
18 - Trade Prizes (never implemented)
19 - Matt/Richard(for drawing the achievements!) - Reward/Icons (done)

Green: acceptance criteria fulfilled
Yellow: acceptance criteria partially fulfilled
Red: feature not implemented

Evolution

In the early stages of the project's development, the design was very simple and lacked an architecture that could provide a high level of modularity and scalability; we simply had a UML diagram that lacked a sophisticated and robust design. We wanted the trait of modularity as this would allow for clean code where the separation of concerns gives us the ability to work more efficiently without many conflicts during development. And we also wanted to take scalability into account for the future possibility of having numerous users. Therefore, after researching into some possible architectures that would allow for these traits, we came across the Model-View-Presenter (MVP) architecture, an advanced form of Model-View-Controller (MVC). MVP is superior to MVC in that the separation of concerns is greater in MVP than MVC as the presenter knows about the model and the view, but the model and view don't know anything about each other. MVC on the other hand has links between the model and view. However, as we began to program with the MVP architecture, it quickly became apparent that the presenters were doing mundane tasks such as binding a model to the view through several function calls. This prompted further research which resulted in the introduction to the use of the MVP with Supervising Controller architecture. In this architecture, the presenters serve to carry out high level logic between the view and model; however, low level tasks such as data binding (model bound to view for display) are done between the model and view, so there is a very weak link between the two. To enforce this concept of data binding, views took in interfaces with only getters, which were implemented by the actual models. Finally, with the addition of the HandlerManager (event bus), it acted as a powerful connector for the entire application and strengthened the separation of concerns as components would know to fire events onto the bus, but only components that cared about the events would react to them, meaning that adding and removing components would be very easy.

Verification

Using FindBugs as our Static Analysis Tool. Running the tool on the project revealed that very few bugs existed in the project; 5 source code ones and 10 tests ones, all of which have been resolved (see pdf containing xml results). These bugs were mostly dead code ones where objects had been initialized but not used anywhere in the code.

Testing

The major benefit of using the MVP model for this project from a testing perspective was the ability to run tests really frequently without having to worry about the high overhead of GWTTestCase tests. Since most of the widget code sits in the view packages, and MVP enforces minimal view code, we end up with a situation where GWTTestCase doesn't have to be utilized as much. Instead, we use vanilla Junit tests for the bulk of our code. This made debugging tests and fixing broken tests (after changes to methods) really easy because standard JRE tests run incredibly fast.

Testing Coverage

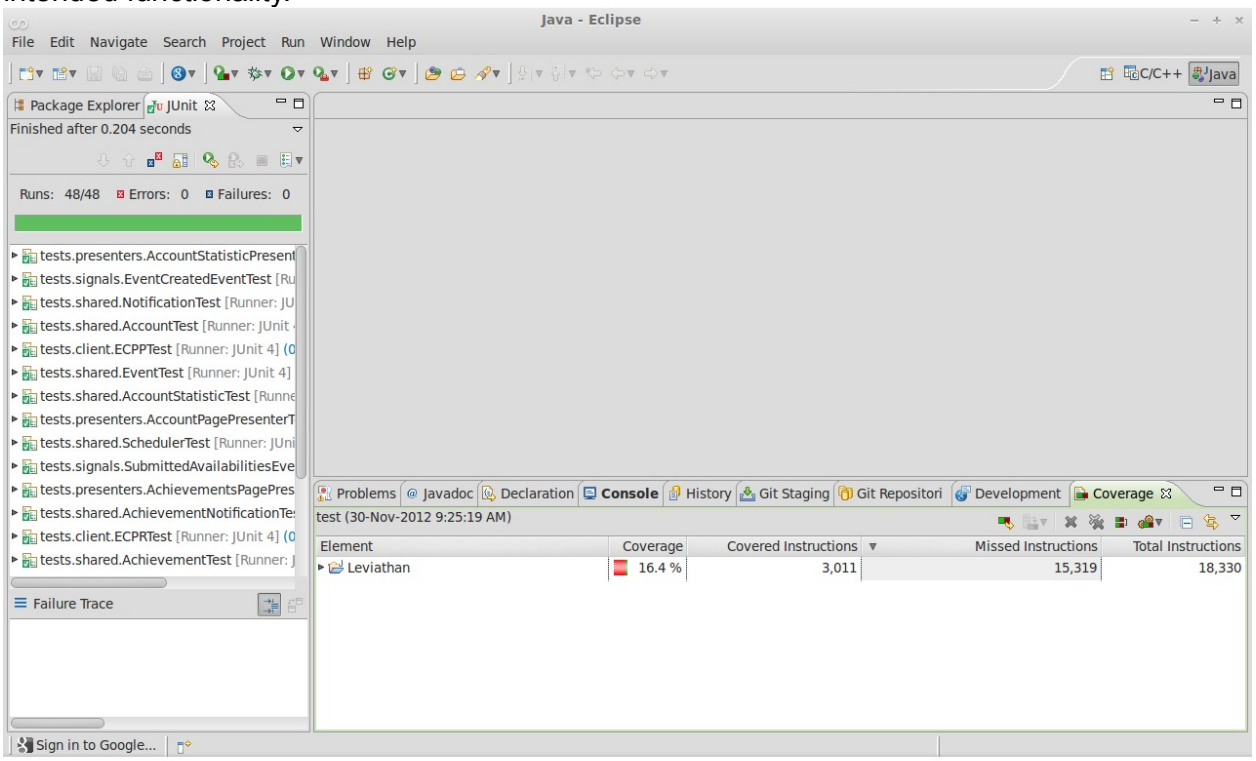
The core modules that are utilized throughout the application (shared package) are thoroughly tested. This is some of the most important code of the application because it gives us our core functionality.

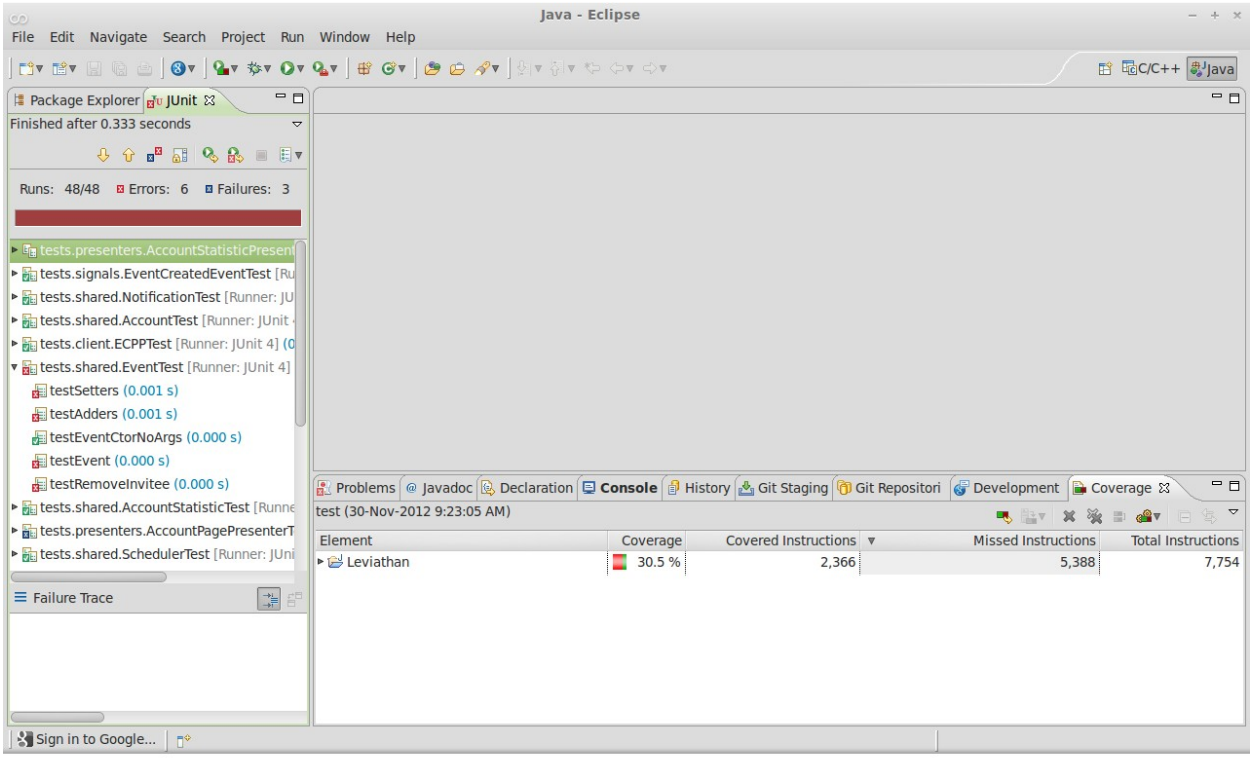
There are parts of the client and presenter packages package that couldn't be tested due to time constraints and the time required to fix existing tests when new code was being added everyday. We used EclEmma to scan through our code for test coverage and here are the results

(screenshots on the next page):

With view code = 16.4%
Without view code = 30.5% + 4.1% due to ECPRTest (0.1) ECPVTest (3.9) EventTest (0.1) breaking when we ran our tests without the view code. So in total, our test coverage is
4.1 + 30.5 = 34.6%

This is considerably lower than the expected 60% mainly due to time constraints which prevented fully testing the presenter, signals and client code. Code was also added during the last couple of days thereby further lowering our coverage. Private methods are also included in this coverage and are highlighted as red and there doesn't seem to be any way to get Emma to ignore these. I did not write tests for any of the view packages because all view modules extend some GWT widget (panels, buttons, labels). The presence of these widgets upon running the application indicates their successful construction. Physical tests (clicking, and other interactions) by us further confirm the intended functionality.





Separation of Concerns

Stuffplotter consists of clients and server. On the server side, we use the Objectify library to communicate with the Google data store and define RPC methods that take full advantage of Java libraries.

In the client packages, we separated the client side of our application into two more packages: presenters and views. Our view classes extend different types of GWT widgets and implements a interface for presenter classes to get access to widget interfaces. For example, some functions that our view classes may implement are getters for elements with click handlers (these implement the GWT class HasClickHandlers).

The actual event binding logic and other types of business logic fell to our presenter classes. We have a presenter class for every major view class. The presenter classes uses the interfaces implemented by the view classes to interact with them such as by passing in data retrieved from the data store for the view classes to display in their widgets.

We also separated authentication for our services by outsourcing that responsibility to an external service. Whenever we need to authenticate, we make an Oauth request that will trigger a pop up prompting the user to log in or grant access. The service will return to us a token in its callback method and we use the token to perform what we need with it. When we are done, we will revoke access to the token so there won't be a window of opportunity for other third parties to use the token.

Security

Stuff Plotter is built using GWT SDK and AppEngine tools, and is built and deployed on Google's AppEngine servers. Most of the third party libraries, such as Maps, Calendar and even OAuth Users

were mainly developed and maintained by Google themselves, so it safe assume most of the common and severe security issues have already been dealt with and are embedded within Google's distribution library. However that being said there are still many security issues that Google could not protect our app from and that we had to ensure additional security ourselves, such as input string checking while adding a friend.

Here is a checklist of the SANS-top 25 errors that our application is protected from.

1. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- Nowhere in our application is there an input box that will generate or call any SQL code

2. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

- Nowhere in our application there is an opportunity to run OS code. This is due to the fact, our application is built and deployed on Google Servers, and all of our client code is run off a web browsers that is only used to communicate with our app and other google's web services.

3. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- Most of the cross-site scripting is protected. There was an issue where the user's login token would appear as part of the URL component, but that issue has been resolved.

4. Unrestricted Upload of File with Dangerous Type
- You cannot upload/download any files using our application

5. Cross-Site Request Forgery (CSRF)
- This is of minor concern and seems to be intended behavior from using Google's User libraries. Using our app requires a login with a Google Account, doing so during that session the user can also access other Google services with that account.

6. URL Redirection to Untrusted Site ('Open Redirect')
- The only redirects are within Google trusted sites, such as a login screen.

7. Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- Written in Java, buffer overflow is not a problem and is checked and compiled using Google API's

8. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
- Our app does not allow access to any path traversal of any kind.

9. Download of Code Without Integrity Check
- Our app does not allow any sort of downloads

10. Inclusion of Functionality from Untrusted Control Sphere
- All users have basic accessibility. There are only 2 accounts with administrative access, but these levels of access are never shared with any other user.

11. Use of Potentially Dangerous Function
- Completely avoided the user of potentially dangerous functions.

12. Incorrect Calculation of Buffer Size
13. Uncontrolled Format String
14. Integer Overflow or Wraparound

- Written and compiled in Java, all buffer/code overflow related warnings/error were taken care of during compile time.That includes checking for null terminated strings.

15. Missing Authentication for Critical Function
16. Missing Authorization
- Stuff Plotter will not render the UI till a valid login with a Google account has taken place

17. Use of Hard-coded Credentials
- There are no hard coded Credentials anywhere in our code.

18. Missing Encryption of Sensitive Data
- There is no missing encryption of sensitive data, and only sensitive data is handled by Google's Oauth user libraries.

19. Reliance on Untrusted Inputs in a Security Decision
- Have rigorously tested. There is only a few times you can input things in our app: Adding friend, creating an event, and creating a comment. Most of these have input checks already in place (for example, you can only add a friend with an "@gmail.com" email address). Also adding a friend ensures no spam is sent to the user as coded in the backend.

20. Execution with Unnecessary Privileges
- We have no code implemented that changes permission of users

21. Incorrect Authorization
- All authorization with a Google Account is handled by using Google's libraries and web services.

22. Incorrect Permission Assignment for Critical Resource
- No users, including the administrative users, have assignment permission with any critical resources.

23. Use of a Broken or Risky Cryptographic Algorithm
24. Improper Restriction of Excessive Authentication Attempts
25. Use of a One-Way Hash without a Salt
The above three are highly dependent on Google's libraries and web services, with an assumption that Google "salts" their password database (for 25).

Possible **Wow Factors** to Consider:
1) Rigorous Javadoc present in most of the files e.g TimeSheetPanel.
2) Custom images and gif for the application.
3) Updated UML diagrams.
4) Richard made the switch to a well tested and documented architecture like MVP after we had one of our weekly meetings.