Project Number: GFP1202

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

____

Christopher Casola

____

Andrew Hurle

____

Jennifer Page

Date: April 25, 2013

Approved:

____

Professor Gary F. Pollice, Major Advisor

1. Software engineering education
2. Modular design
3. Java

Abstract

This Major Qualifying Project involved designing and writing a client and exemplar module for the WPI Suite TNG core server. Our client, named Janeway, was written in Java and provides module developers with a standard method for interacting with the server and the user. The goal for Janeway was to enable software engineering students to accomplish these tasks without needing much knowledge of network protocols or languages other than Java. Our goal for the exemplar module was to provide a useful example for students to reference when building a module for WPI Suite TNG. We also wrote a significant amount of developer documentation to assist students who needed to set up their development environment or use the various software APIs that the exemplar and core teams provided.

Acknowledgements

First, we would like to thank Professor Gary Pollice for his full support and guidance throughout this project. His feedback on our project's goals and progress was a great contribution to our success. The help that he provided in synchronizing priorities between the Core project team and our team was invaluable.

We would also like to thank the WPI Suite Core project team for being an excellent group to work with throughout the project. Their ability to collaborate with us on the Core server module API and the REST API ensured that our combined projects were a success and ready for students to use in D term.

Lastly, we would like to thank the D term 2013 software engineering students for being the first class to develop for WPI Suite TNG. Thank you for being patient when the inevitable issues arose and providing valuable feedback that we can use to improve WPI Suite in the future. Finally, thank you for contributing a new Requirements Manager to WPI Suite!

Table of Contents

List of Figures

List of Tables

1 Introduction

Exposing students to a large software project in an introductory software engineering course, while at the same time teaching basic design principles, can be extremely challenging. It is difficult to find a project of optimal size, where it is large enough to be usefully complex but small enough for a beginner to comprehend and contribute to within the short time frame of a college course. A suitable project is one that not only employs good design principles and is well documented, but that is easily extensible and modular. This modularity gives students the opportunity to extend the functionality of the software in small, manageable projects without needing to worry about fully understanding the core of the existing application.

In a bid to provide a suitable project, Professor Gary Pollice of Worcester Polytechnic Institute (WPI) began working with MQP teams to develop an application called WPI Suite. As the project evolved, it was used in a few of WPI's software engineering course offerings, leading to the development of many modules by students. The project has worked well as an education tool, but many of the modules existing today are of varying quality. The monolithic structure of the application, where all logic is contained in the client and each client connects directly to a shared database, is causing design, security, and performance problems.

Another MQP team has designed and implemented a new WPI Suite core using a client-server architecture that provides user and project management, persistence, and a stateless network abstraction to modules. To accompany the new core, the exemplar project team has developed a defect-tracking module to serve as an example for others. The exemplar module includes extensive documentation to help students who are tasked with developing their own modules. The goal of the module is to get students started quickly by providing clear example code that they can use as their guide during the development process. In addition to the module, the project team has also provided a desktop client for WPI Suite that includes a Java-based network library to simplify the process of making calls to the core API.

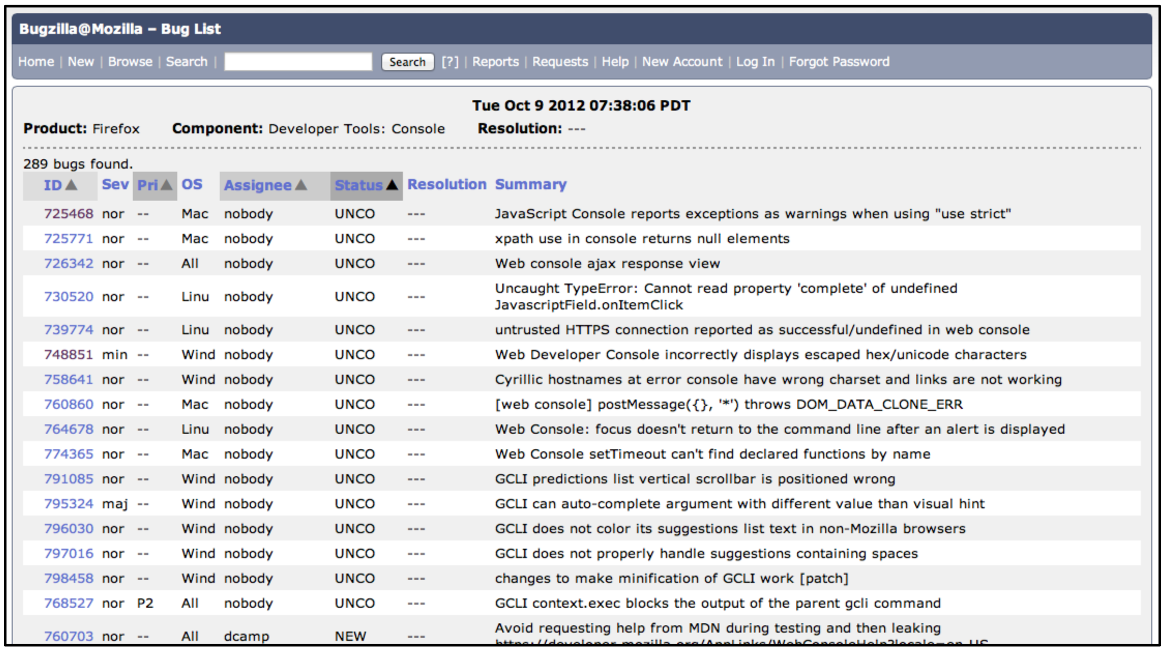
The remainder of this document contains an overview of existing defect trackers and the features they provide, a discussion of the methodology used to manage and complete this project, results showing the success of WPI Suite as an educational tool, and recommendations for future work.

2 Background

There are many defect tracking tools available and they range from powerful and customizable, to simple and immutable. In order to create an informed design for the WPI Suite defect tracking module, the team performed an analysis of existing tools. The team paid particular attention to the usability and feature sets of the trackers, as well as how well they were regarded in the software community. The tools that were reviewed include: Bugzilla, FusionForge Tracker, GitHub Issues, JIRA, and Trac.

2.1 Bugzilla

Bugzilla is a free and open source defect tracking system initially developed for Mozilla. It is now used by many active projects, including Mozilla's Firefox, the Linux Kernel, Eclipse, and the Apache Project (Bugzilla, 2012). Bugzilla organizes bugs by Product and Component. Each bug has many fields including assignee, subject, and priority. When the bug is initially created a first comment is included, then other users can add additional comments and attach files (e.g. patches, screenshots). The tool also has built-in support for patch review – the author of a patch can request that another user review a patch. The reviewer is then notified by email, demonstrating Bugzilla's tight backend email integration. The reviewer can view the patch and add comments line-by-line. Users can also directly add email addresses to the CC field on a bug so that others are notified of changes and comments. Users can use the search or browse features to view bugs. Both options result in a large table of bugs, each of which is clickable to see details and comments (see Figure 1).



Bugzilla@Mozilla – Bug List

Home | New | Browse | Search | Search [?] | Reports | Requests | Help | New Account | Log In | Forgot Password

Tue Oct 9 2012 07:38:06 PDT

Product: Firefox **Component:** Developer Tools: Console **Resolution:** ---

289 bugs found.

ID▲	Sev	Pri▲	OS	Assignee▲	Status▲	Resolution	Summary
725468	nor	--	Mac	nobody	UNCO	---	JavaScript Console reports exceptions as warnings when using "use strict"
725771	nor	--	Mac	nobody	UNCO	---	xpath use in console returns null elements
726342	nor	--	All	nobody	UNCO	---	Web console ajax response view
730520	nor	--	Linu	nobody	UNCO	---	Uncaught TypeError: Cannot read property 'complete' of undefined JavascriptField.onItemClick
739774	nor	--	Linu	nobody	UNCO	---	untrusted HTTPS connection reported as successful/undefined in web console
748851	min	--	Wind	nobody	UNCO	---	Web Developer Console incorrectly displays escaped hex/unicode characters
758641	nor	--	Wind	nobody	UNCO	---	Cyrillic hostnames at error console have wrong charset and links are not working
760860	nor	--	Mac	nobody	UNCO	---	[web console] postMessage({}, '*') throws DOM_DATA_CLONE_ERR
764678	nor	--	Linu	nobody	UNCO	---	Web Console: focus doesn't return to the command line after an alert is displayed
774365	nor	--	Mac	nobody	UNCO	---	Web Console setTimeout can't find declared functions by name
791085	nor	--	Wind	nobody	UNCO	---	GCLI predictions list vertical scrollbar is positioned wrong
795324	maj	--	Wind	nobody	UNCO	---	GCLI can auto-complete argument with different value than visual hint
796030	nor	--	Wind	nobody	UNCO	---	GCLI does not color its suggestions list text in non-Mozilla browsers
797016	nor	--	Wind	nobody	UNCO	---	GCLI does not properly handle suggestions containing spaces
798458	nor	--	Wind	nobody	UNCO	---	changes to make minification of GCLI work [patch]
768527	nor	P2	All	nobody	UNCO	---	GCLI context.exec blocks the output of the parent gcli command
760703	nor	--	All	dcamp	NEW	---	Avoid requesting help from MDN during testing and then leaking

Figure 1: Bugzilla User Interface

Limited customization of Bugzilla (e.g. adding custom fields) can be accomplished using the Web interface. More advanced configuration must be performed by editing configuration files, Perl scripts, templates, or CSS files. Table 1 displays an analyses of the pros and cons of Bugzilla’s feature set.

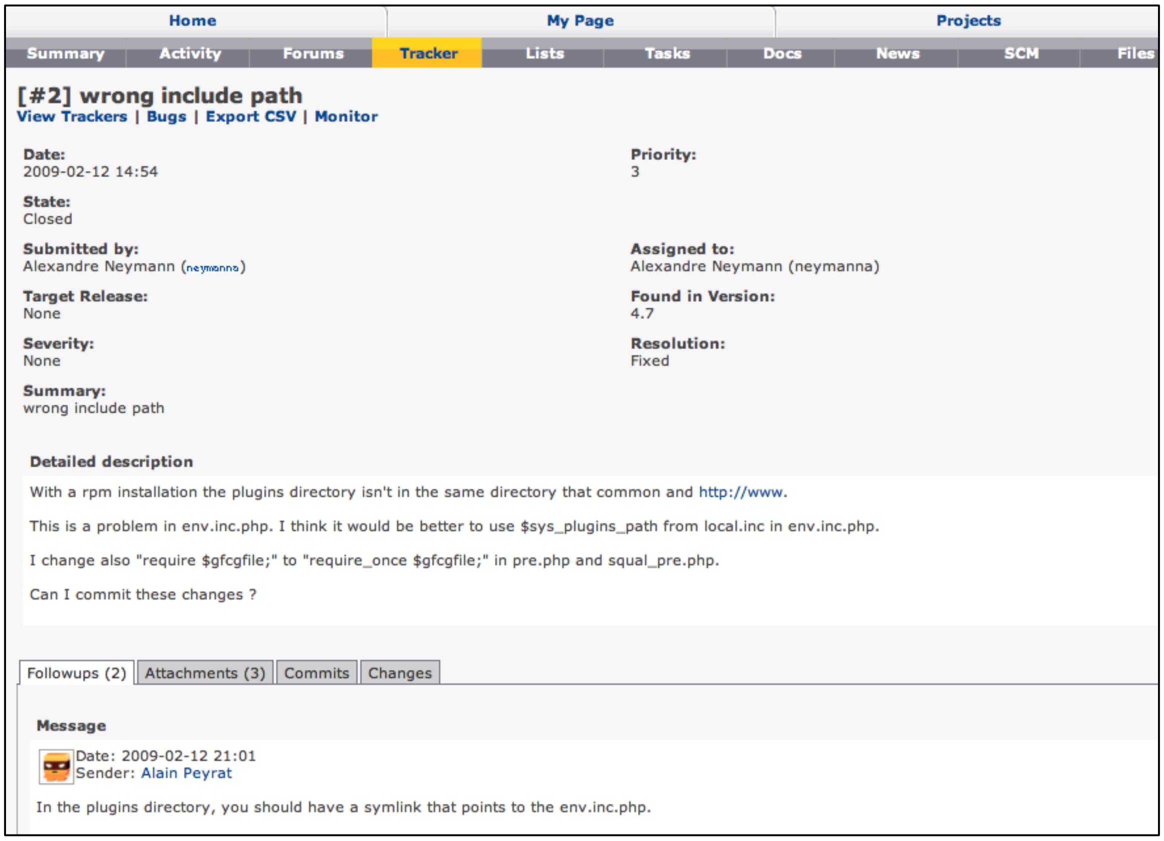
Table 1: Evaluation of Bugzilla

Pros	Cons
Powerful and customizable	The interface is cluttered and not optimally formatted (e.g. important fields are not highlighted, too many fields are displayed)
Proven in the field and scales well (hundreds of thousands of bugs)	Performance issues when listing many bugs
Tight email integration	User preferences can be overwhelming, and the defaults may be unreasonable (e.g. getting emails about every bug)
Extensive control over user preferences	Permissions system may not be clear to a normal user
Built-in patch review	You have to read through all comments to learn the state of a bug
Support for bug dependencies, can display a graph/tree of dependent bugs	

2.2 FusionForge Tracker

Tracker is a generic tracking module included with FusionForge. It can be used to track anything that the project needs, including bugs, feature requests, and patches. In fact, the FusionForge project tracks the previously mentioned items, but additional trackers can be added for other purposes.

The interface is relatively simple. When Tracker is opened a list of tracked items is displayed and a tab bar is shown at the top with various search and filter options. As opposed to Bugzilla, filter options are selected by adding fields that the user wants to use as criteria, instead of being presented with every possible option. Once the user opens a bug, another set of tabs is displayed beneath the bug field and description that lets users easily see follow-ups, attachments, commits, and changes (see Figure 2). An interesting feature unique to Tracker is canned responses. These pre-composed messaged can be used when commenting on bugs, but are most useful for public ticketing systems.



[#2] wrong include path

View Trackers | Bugs | Export CSV | Monitor

Date: 2009-02-12 14:54	**Priority:** 3
State: Closed	
Submitted by: Alexandre Neymann (neymanna)	**Assigned to:** Alexandre Neymann (neymanna)
Target Release: None	**Found in Version:** 4.7
Severity: None	**Resolution:** Fixed
Summary: wrong include path	

Detailed description

With a rpm installation the plugins directory isn't in the same directory that common and http://www.

This is a problem in env.inc.php. I think it would be better to use $sys_plugins_path from local.inc in env.inc.php.

I change also "require $gfcgfile;" to "require_once $gfcgfile;" in pre.php and squal_pre.php.

Can I commit these changes ?

Message

Date: 2009-02-12 21:01
Sender: Alain Peyrat

In the plugins directory, you should have a symlink that points to the env.inc.php.

Figure 2: FusionForge Tracker User Interface

On the administrative side, there is a web interface where admins can add custom fields and control notification settings. With Tracker, users cannot set their notification settings as easily as they can with Bugzilla. Tracker administrators set most of the notification settings by controlling the mailing lists within the admin interface. Lastly, there is a command line interface for interacting with bugs that may be useful for certain working environments. Table 2 displays a summary of the pros and cons for FusionForge Tracker.

Table 2: Evaluation of FusionForge Tracker

Pros	**Cons**
Easy to reach search – field-adding paradigm easier to work with than other trackers	Changes are shown on a different tab than comments making it harder to see the progression over time
Canned responses very useful for certain projects	No support for defect dependencies
User avatars	

2.3 GitHub Issues

The defect tracking module provided by GitHub is called GitHub Issues. It is available via the GitHub Web application and is enabled by default on all repositories. The feature set of Issues is small compared to other defect trackers, but it is powerful because of its integration with the project's Git repository.

To create an issue, the user provides a title and brief description. They can then assign a person to the issue and apply predefined labels. One of the unique features of Issues is the ability to enter markdown in the description field. GitHub's custom markdown language, called *GitHub Flavored Markdown*, enables users to easily include code snippets and other formatting in the issue description. The description editor has two tabs: one to enter the markdown and another that provides an instant preview.

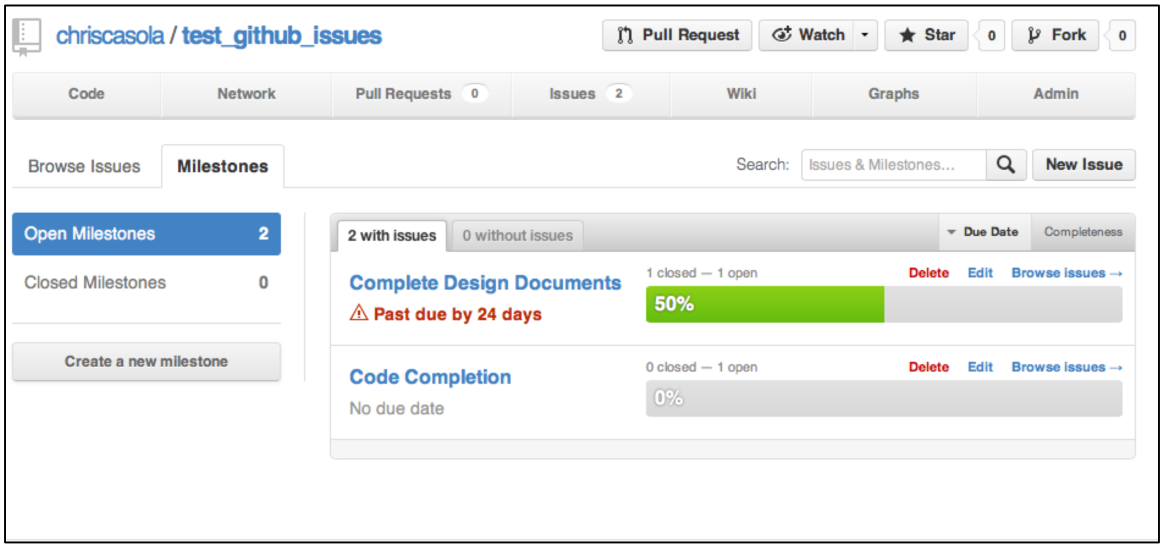


Figure 3: GitHub Issues - Milestones

In GitHub, issues are organized by milestone. A milestone consists of a due date and a list of assigned items. GitHub provides a view that shows the progress being made on each milestone using graphical progress bars (see Figure 3). As issues assigned to a milestone are closed, the completion level automatically increases.

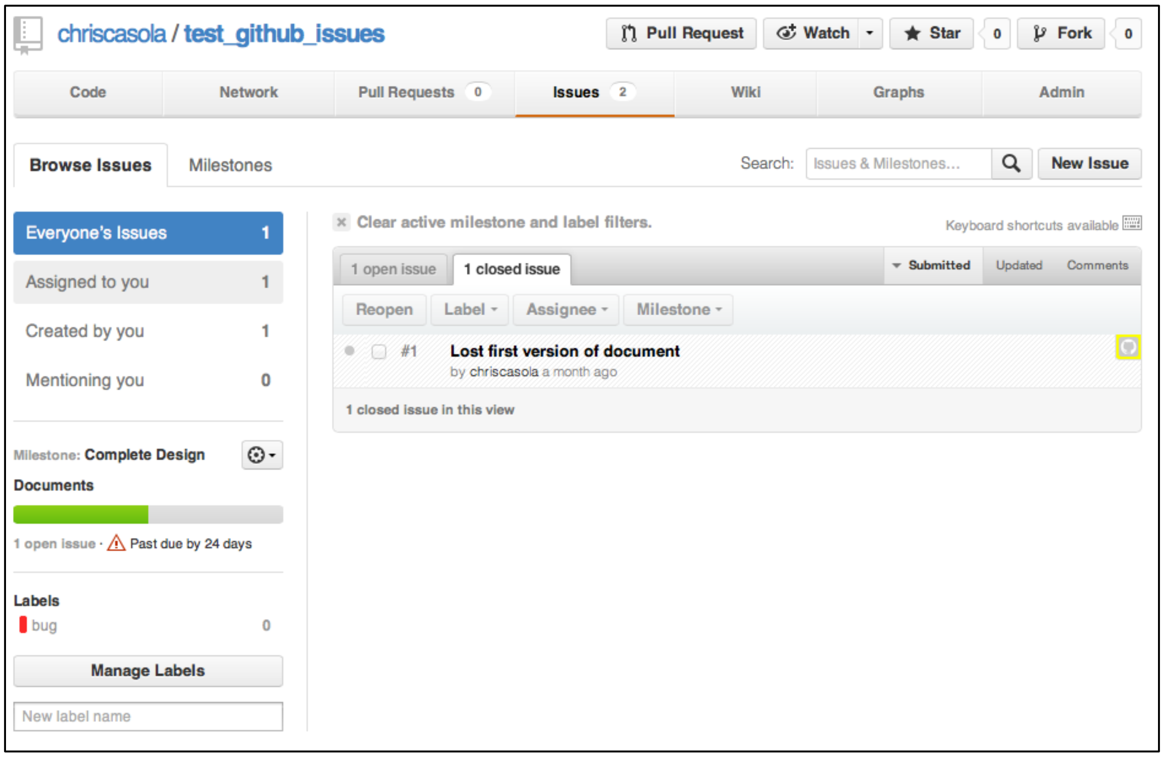


Figure 4: GitHub Issues User Interface

The Issues user interface allows users to filter issues by assignee, creator, and whether or not the current user has been mentioned in the issues (see Figure 4). Once the user clicks on an issue, they can view the comment thread and add their own comments. An issue can be closed either by clicking the close button in the browser, or by including the text '`closes GH-[issue #]`' in a commit comment.

More advanced defect tracking features are where GitHub falls short. It does not offer any type of time tracking ability or automatic issue assignment. It also does not allow multiple users to be assigned to the same issue.

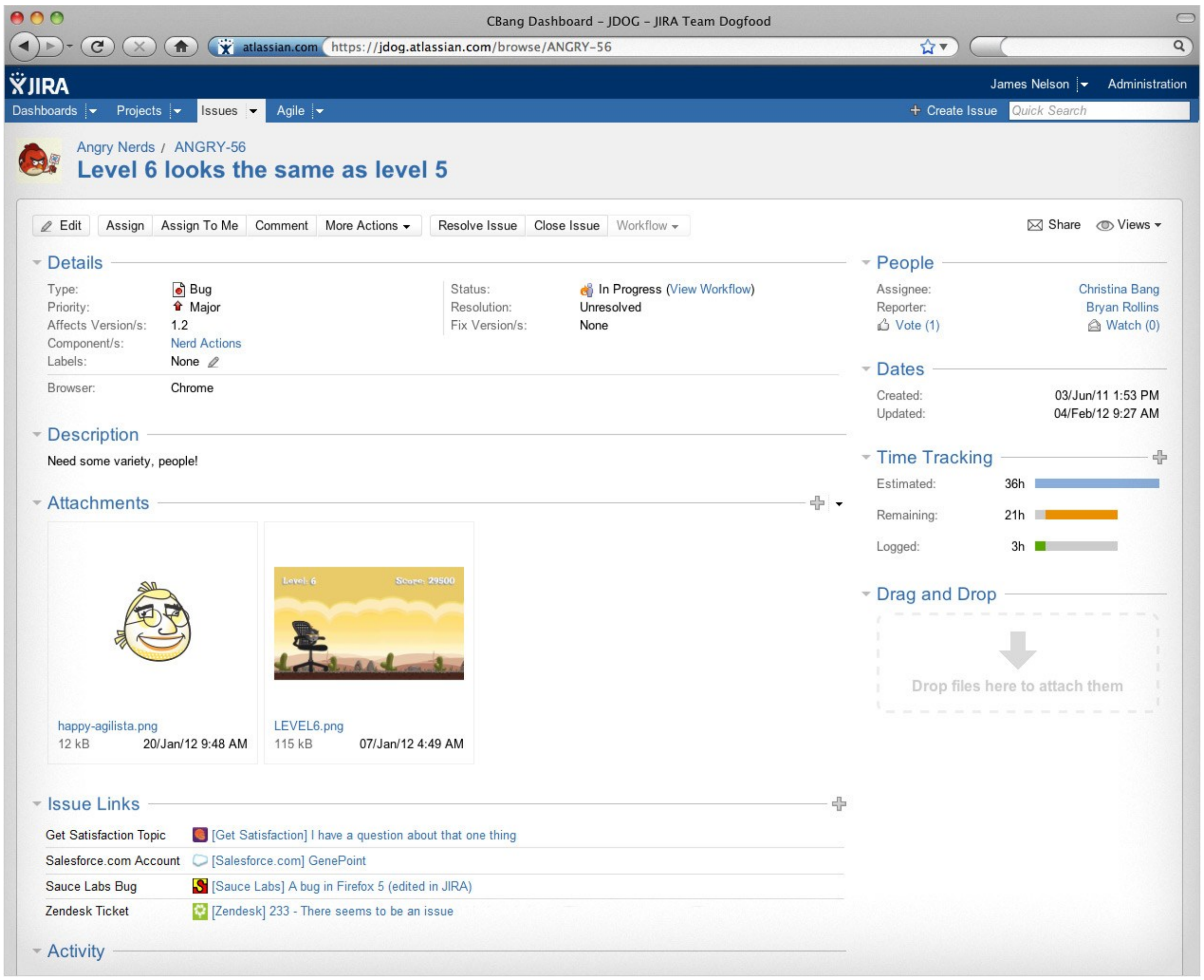
Table 3: Evaluation of GitHub Issues

Pros	**Cons**
Tight integration with Git repositories	No time-tracking ability
Ability to enter markdown in text fields to describe formatting and syntax highlighting	No automatic issues assignment
Milestones allow issues to be grouped and overall progress to be tracked	No custom fields
Can close issues by specifying certain text in commit messages	Multiple users cannot be assigned to the same issue

2.4 JIRA

JIRA is a commercial defect tracker and project management tool developed by Atlassian. It is implemented in Java. Its Web interface is the primary means for interaction with users, but it also supports input via email, CLIs, a local client, REST and SOAP APIs, Mylyn (Eclipse), and Visual Studio. JIRA can also provide notifications via email, XMPP, and RSS. Version control systems supported by JIRA include CVS, Git, Mercurial, and Subversion. Multiple SQL back-ends are supported by JIRA when self-hosted. A cloud based JIRA service is also available.

Issue creation in JIRA requires a user to input required fields. Optional fields are also available. The required fields as well as the defaults for the fields are customizable on a per project basis. This allows for simplification of issue submission, as well as the ability to ensure that the correct information is filled out. Issues may be commented on in order to show progress and troubleshoot the issue. Closing the issue is a fairly simple process involving filling out a few required fields. The requirements for this process may also be customized by administrators. Figure 5 shows a view of an issue in JIRA's web interface (Atlassian, 2012).



Angry Nerds / ANGRY-56

Level 6 looks the same as level 5

Details

Type:	Bug	Status:	In Progress (View Workflow)
Priority:	Major	Resolution:	Unresolved
Affects Version/s:	1.2	Fix Version/s:	None
Component/s:	Nerd Actions		
Labels:	None		
Browser:	Chrome		

Description

Need some variety, people!

Attachments

happy-agilista.png 12 kB 20/Jan/12 9:48 AM

LEVEL6.png 115 kB 07/Jan/12 4:49 AM

Issue Links

Get Satisfaction Topic	[Get Satisfaction] I have a question about that one thing
Salesforce.com Account	[Salesforce.com] GenePoint
Sauce Labs Bug	[Sauce Labs] A bug in Firefox 5 (edited in JIRA)
Zendesk Ticket	[Zendesk] 233 - There seems to be an issue

Activity

People

Assignee:	Christina Bang
Reporter:	Bryan Rollins
Vote (1)	Watch (0)

Dates

Created:	03/Jun/11 1:53 PM
Updated:	04/Feb/12 9:27 AM

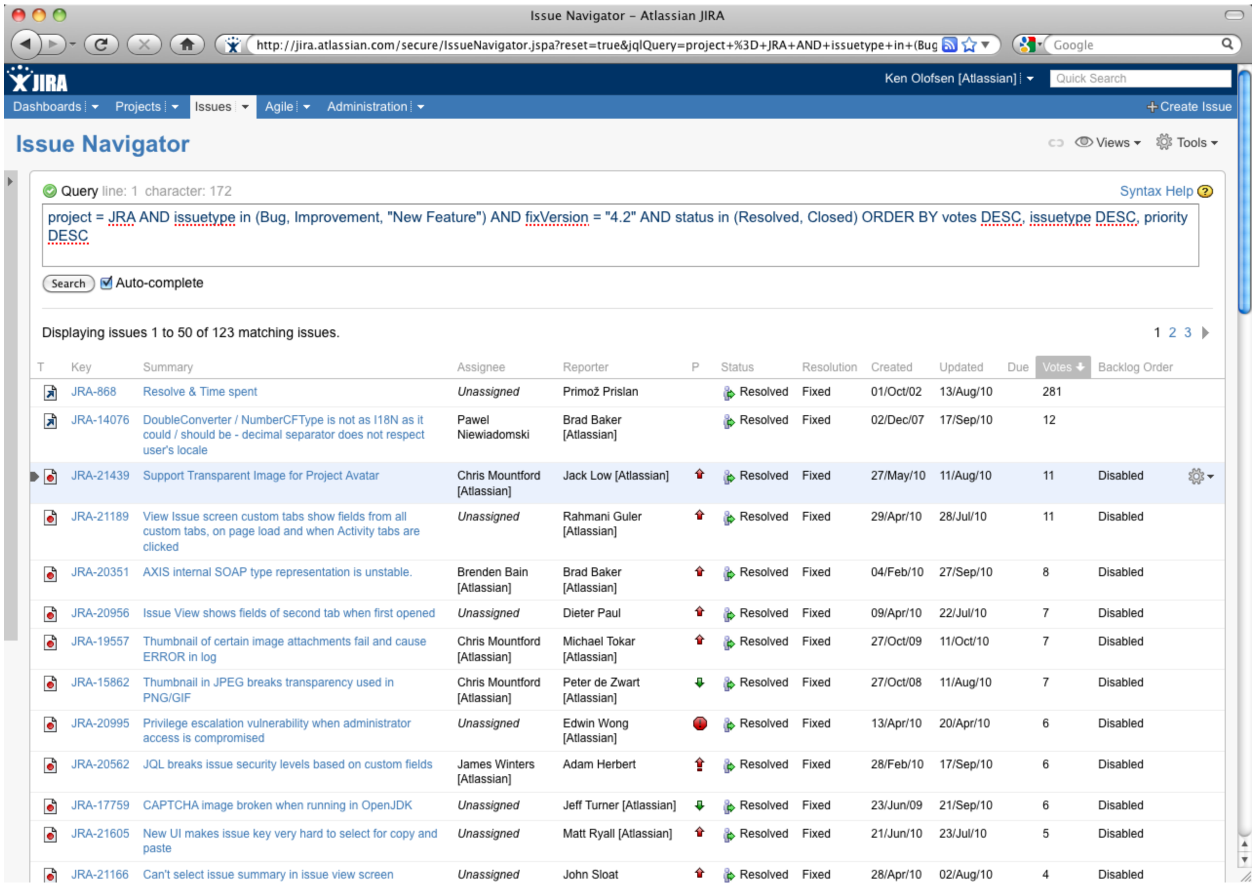
Time Tracking

Estimated:	36h
Remaining:	21h
Logged:	3h

Drag and Drop

Figure 5: JIRA - Viewing an issue through the web interface

One of JIRA's most powerful features is JQL, JIRA's SQL-esque query language. JQL allows a user to perform custom and complex search queries for issues in JIRA. The results returned can be viewed as a single list of issues or can be incorporated into JIRA's dashboards in the form of lists and charts. Figure 6 shows the Issue Navigator, which is displaying the results of a JQL query (Atlassian, 2012). The JQL query can be seen in the top quarter of the webpage.


Query line: 1 character: 172

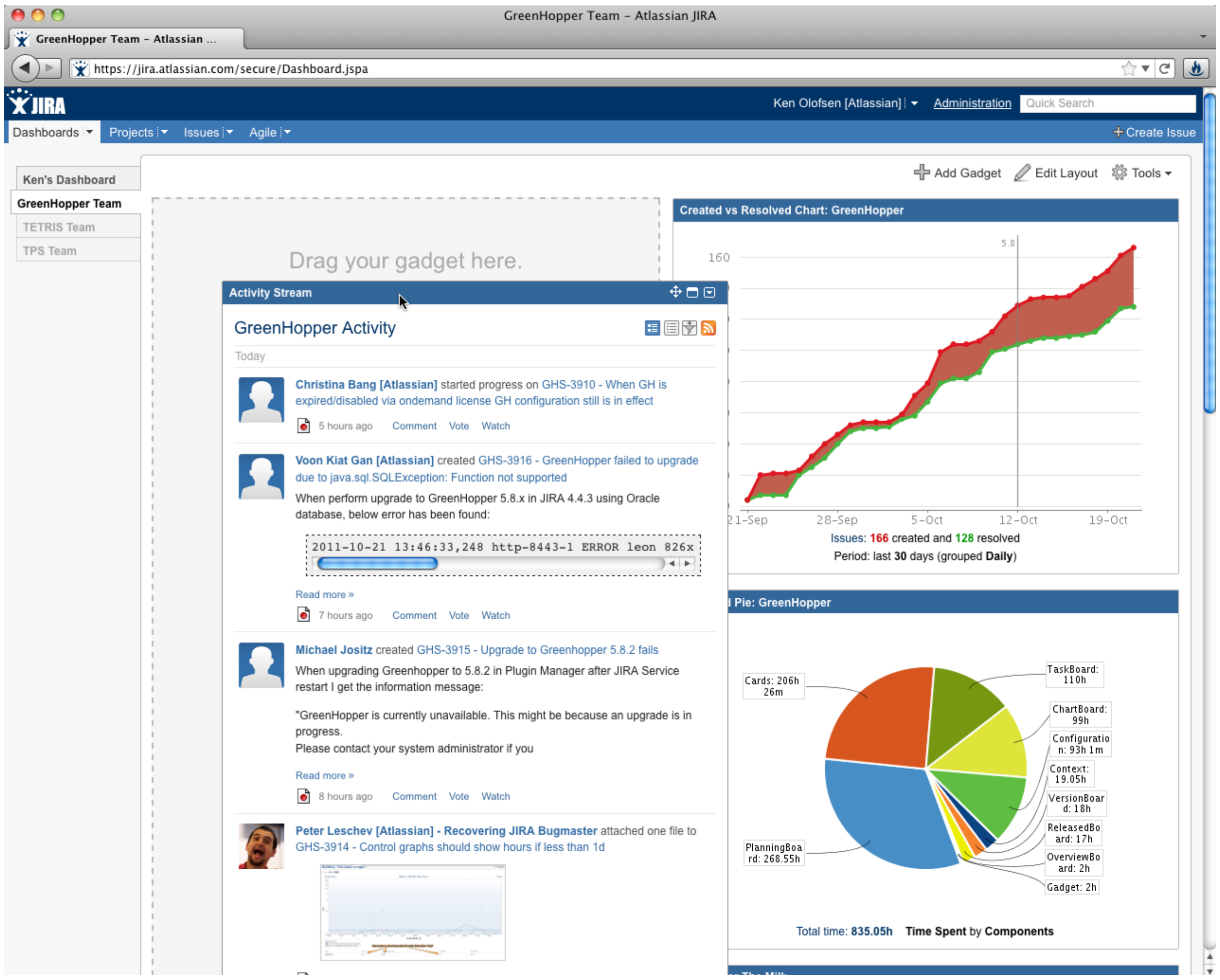
project = JRA AND issuetype in (Bug, Improvement, "New Feature") AND fixVersion = "4.2" AND status in (Resolved, Closed) ORDER BY votes DESC, issuetype DESC, priority DESC

Displaying issues 1 to 50 of 123 matching issues.

T	Key	Summary	Assignee	Reporter	P	Status	Resolution	Created	Updated	Due	Votes	Backlog Order
	JRA-868	Resolve & Time spent	*Unassigned*	Primož Prislan		Resolved	Fixed	01/Oct/02	13/Aug/10		281	
	JRA-14076	DoubleConverter / NumberCFType is not as I18N as it could / should be - decimal separator does not respect user's locale	Pawel Niewiadomski	Brad Baker [Atlassian]		Resolved	Fixed	02/Dec/07	17/Sep/10		12	
	JRA-21439	Support Transparent Image for Project Avatar	Chris Mountford [Atlassian]	Jack Low [Atlassian]	⇧	Resolved	Fixed	27/May/10	11/Aug/10		11	Disabled
	JRA-21189	View Issue screen custom tabs show fields from all custom tabs, on page load and when Activity tabs are clicked	*Unassigned*	Rahmani Guler [Atlassian]	⇧	Resolved	Fixed	29/Apr/10	28/Jul/10		11	Disabled
	JRA-20351	AXIS internal SOAP type representation is unstable.	Brenden Bain [Atlassian]	Brad Baker [Atlassian]	⇧	Resolved	Fixed	04/Feb/10	27/Sep/10		8	Disabled
	JRA-20956	Issue View shows fields of second tab when first opened	*Unassigned*	Dieter Paul	⇧	Resolved	Fixed	09/Apr/10	22/Jul/10		7	Disabled
	JRA-19557	Thumbnail of certain image attachments fail and cause ERROR in log	Chris Mountford [Atlassian]	Michael Tokar [Atlassian]	⇧	Resolved	Fixed	27/Oct/09	11/Oct/10		7	Disabled
	JRA-15862	Thumbnail in JPEG breaks transparency used in PNG/GIF	Chris Mountford [Atlassian]	Peter de Zwart [Atlassian]	⇩	Resolved	Fixed	27/Oct/08	11/Aug/10		7	Disabled
	JRA-20995	Privilege escalation vulnerability when administrator access is compromised	*Unassigned*	Edwin Wong [Atlassian]	●	Resolved	Fixed	13/Apr/10	20/Apr/10		6	Disabled
	JRA-20562	JQL breaks issue security levels based on custom fields	James Winters [Atlassian]	Adam Herbert	⇧	Resolved	Fixed	28/Feb/10	17/Sep/10		6	Disabled
	JRA-17759	CAPTCHA image broken when running in OpenJDK	*Unassigned*	Jeff Turner [Atlassian]	⇩	Resolved	Fixed	23/Jun/09	21/Sep/10		6	Disabled
	JRA-21605	New UI makes issue key very hard to select for copy and paste	*Unassigned*	Matt Ryall [Atlassian]	⇧	Resolved	Fixed	21/Jun/10	23/Jul/10		5	Disabled
	JRA-21166	Can't select issue summary in issue view screen	*Unassigned*	John Sloat	⇧	Resolved	Fixed	28/Apr/10	02/Aug/10		4	Disabled

Figure 6: JIRA - JQL results displayed in the Issue Navigator

Dashboards in JIRA allow for any user to create a custom overview of a project or of their responsibilities, among other things. This is useful for a project manager who wants to see how many issues are being created versus resolved over time. A team member might want to view all issues assigned to him or her over multiple projects. Someone in the Quality Assurance group may want to quickly check on the status of issues they have submitted regarding faults in software they are testing. Any dashboard may be shared with other users of the same JIRA instance. Figure 7 shows a dashboard in the process of being created.



GreenHopper Activity

Today

Christina Bang [Atlassian] started progress on GHS-3910 - When GH is expired/disabled via ondemand license GH configuration still is in effect

5 hours ago Comment Vote Watch

Voon Kiat Gan [Atlassian] created GHS-3916 - GreenHopper failed to upgrade due to java.sql.SQLException: Function not supported

When perform upgrade to GreenHopper 5.8.x in JIRA 4.4.3 using Oracle database, below error has been found:

```
2011-10-21 13:46:33,248 http-8443-1 ERROR leon 826x
```

Read more »

7 hours ago Comment Vote Watch

Michael Jositz created GHS-3915 - Upgrade to Greenhopper 5.8.2 fails

When upgrading Greenhopper to 5.8.2 in Plugin Manager after JIRA Service restart I get the information message:

"GreenHopper is currently unavailable. This might be because an upgrade is in progress.
Please contact your system administrator if you

Read more »

8 hours ago Comment Vote Watch

Peter Leschev [Atlassian] - Recovering JIRA Bugmaster attached one file to GHS-3914 - Control graphs should show hours if less than 1d

Issues: 166 created and 128 resolved
Period: last 30 days (grouped Daily)

Total time: 835.05h Time Spent by Components

Figure 7: JIRA - A dashboard being customized by a user

JIRA's large feature set and relative ease of use has made it popular in corporate environments. Its support of multiple SQL back ends and multiple revision control systems increases the likelihood of compliance with corporate policies and preexisting software licenses. Its compatibility with LDAP authentication allows it to integrate with Windows domains, which eases administrative headaches when dealing with user accounts. The heavy customizability of JIRA makes it easy to enforce various standards for project issue tracking. It also allows for better management of issues and teams via its dashboard and other features.

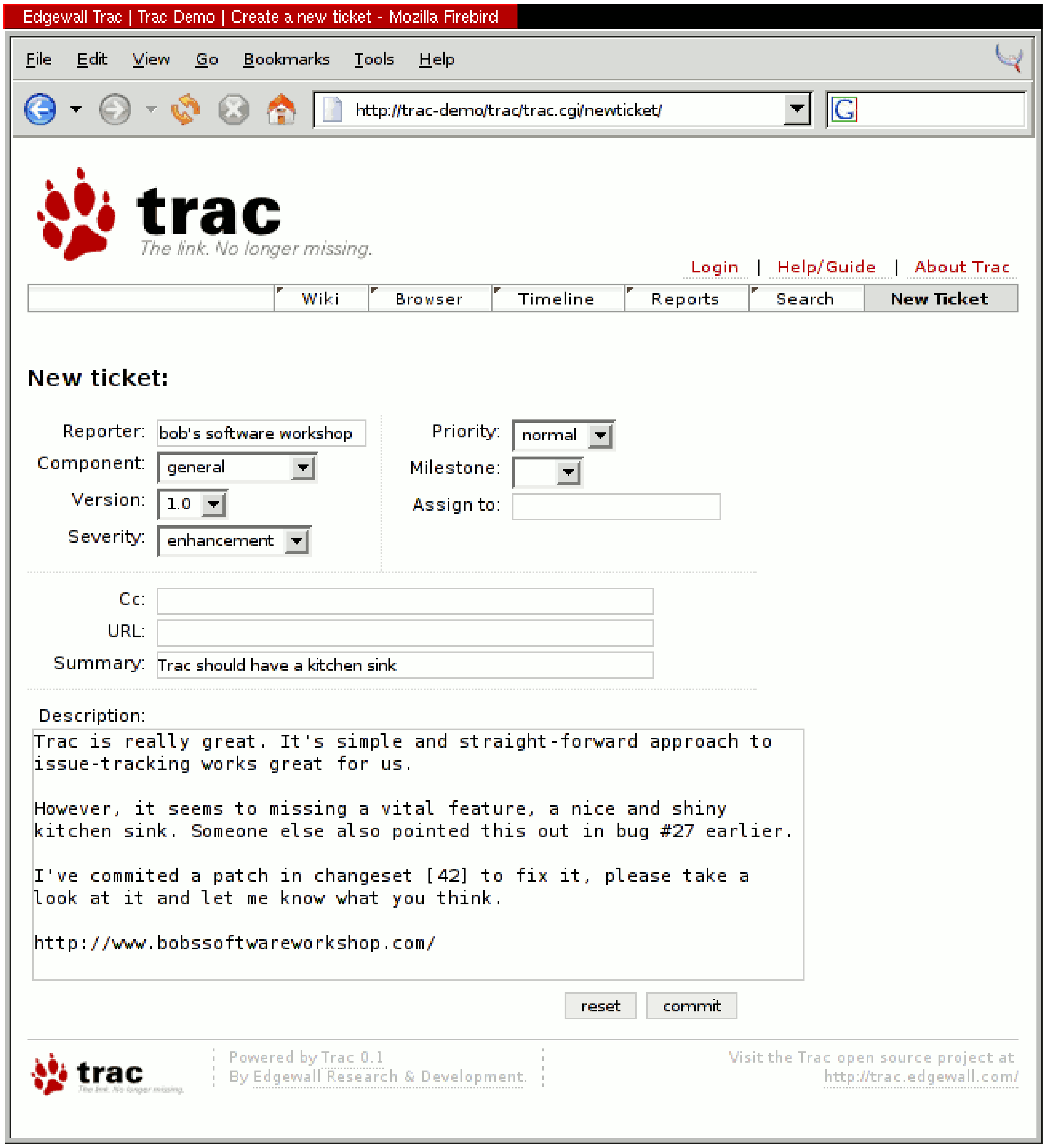
Table 4: Evaluation of JIRA

Pros	**Cons**
LDAP authentication support	No support for OpenID, OAuth, or public key authentication
Supports multiple revision control systems	No two-factor authentication support
Free licenses for open source projects and certain organizations	Proprietary
JQL	
Customizable	
Supports multiple SQL back-ends	

2.5 Trac

Trac is similar to Bugzilla in many ways. Tickets have a similar structure, with similar fields by default and it integrates with email in a similar way. Trac is more user friendly than Bugzilla.

In Bugzilla, the user may edit a ticket while viewing it. Unlike Bugzilla, Trac requires the user to navigate to a separate form for ticket modification. When creating a ticket in Trac, a field exists for entering a description (see Figure 8). In Bugzilla, a *first comment* is used in lieu of a description field. Trac has a simple attachment system without support for review. Comments in Trac allow Wiki markup to be used.


Edgewall Trac | Trac Demo | Create a new ticket - Mozilla Firebird
File Edit View Go Bookmarks Tools Help
http://trac-demo/trac/trac.cgi/newticket/
trac
The link. No longer missing.
Login | Help/Guide | About Trac
Wiki Browser Timeline Reports Search New Ticket

New ticket:

Reporter:	bob's software workshop	Priority:	normal
Component:	general	Milestone:	
Version:	1.0	Assign to:	
Severity:	enhancement		

Cc:	
URL:	
Summary:	Trac should have a kitchen sink

Description:

```
Trac is really great. It's simple and straight-forward approach to
issue-tracking works great for us.

However, it seems to missing a vital feature, a nice and shiny
kitchen sink. Someone else also pointed this out in bug #27 earlier.

I've commited a patch in changeset [42] to fix it, please take a
look at it and let me know what you think.

http://www.bobssoftwareworkshop.com/
```

reset
commit
Powered by Trac 0.1
By Edgewall Research & Development.
Visit the Trac open source project at
http://trac.edgewall.com/

Figure 8: Trac - Ticket creation (Internet Archive, 2011)

One of Trac's features is the ability to view progress on certain milestones. Plugins may be used to customize Trac's abilities. There is also support for report generation. Custom fields are available in

Trac, but unfortunately they may only be added by editing an initialization (INI) file. Trac is widely used by many large software projects, which has helped prove its ability to get its job done.

Table 5: Evaluation of Trac

Pros	**Cons**
Wiki formatting	No native dependency support
More user-friendly than Bugzilla	No patch reviewing support
Roadmap feature	Search is very bare-bones (keyword search only)
Nice UI features like threaded comments view, collapsible areas of bugs	

3 Methodology

This section describes the tools and techniques used, and major design decision made, during the development of the WPI Suite Exemplar Module. First, the project management strategy is discussed, including the team's development methodology and the collaboration tools used to coordinate the project. Next, an overview of the technologies used to implement the product is provided as well as the reasoning behind those choices. Following that is a description of the architecture of the final product and an outline of the features it includes. Last is a discussion of the issues that were encountered during the design and implementation of the project.

3.1 Project Management Strategy

Careful project management was required for this MQP because two teams were working closely to develop software. Documentation, design decisions, user stories, code, and known defects needed to be easily accessible and maintained throughout the life of the project. A number of tools were used to accomplish this including PivotalTracker for requirements management and GitHub for source control and documentation. The process used by the two teams was based on Agile methods with a strong emphasis on generating complete and well-written documentation. Features were implemented on a weekly basis and requirements were determined and changed during the development stage. New features were merged into a stable master branch each week. Rapid team communication was made possible using Internet Relay Chat (IRC).

3.1.1 Iterative Development

Both the Core and Exemplar teams developed code separately. In order to synchronize this code, it was decided that both teams would merge functional code into the `dev` branch every two weeks – the length of one iteration. The two teams staggered their iterations by one week in order to allow time for the Exemplar team to adjust to changes and code deprecation in the Core.

Later, both teams switched to one-week iterations to save time correcting *bit rot* problems, since changes in core interfaces could not be honored in the exemplar module until two weeks later. After making this change, the Exemplar team was able to adapt more quickly to changes in the core and newly created code could utilize more recent core features instead of old or deprecated ones. The Exemplar team took responsibility for merging code from the `dev-core` branch into the `dev-exemplar` branch, and then into the `dev` branch when iterations ended. This change made the Exemplar team the custodians of the `dev` branch and also allowed us to ensure that the code on the `dev` branch was always stable.

3.1.2 Code Review

In order to ensure high code quality, the Exemplar team instituted an informal code review process. Contributors would work on features on separate branches, and when they were done they would merge back into the main development branch. Before doing so, the contributor would generally request a review through IRC or email. The reviewer would look over the changes introduced in the branch. GitHub made reviewing changes simple because the person conducting the review could simply click on the commit in the Network Graph and a diff would be shown indicating where changes were made. The reviewer would also perform some basic quality assurance including checking for poorly documented code, suggesting refactoring of unclear code, ensuring feature completeness, and testing for regressions. If there were problems with the branch, the reviewer and contributor would collaborate to fix these problems, and then the branch would get merged into the main development branch.

Code reviews are an established way to reduce defects and share knowledge among developers. A 2009 study conducted by Mantyla and Lassenius found that “code reviews seem particularly valuable for software product or service businesses in which the same software or service is modified and extended over time” (Mantyla & Lassenius, 2009). Since WPI Suite is intended to be extended over time by developers with varying skill levels, the WPI Suite project relies on code review to help maintain high code quality.

We decided to use an informal and lightweight process for a few reasons. We only have three developers on the exemplar module team. A strict, thorough, and formal review process would hinder our ability to push out a working product in time for students to use it in the last term of the same academic year. Our small team can use their best judgment to decide if a patch is worth reviewing. Secondly, as a Greenfield project (a project not bound by the constraints of prior work) much of our work is done to develop solutions from scratch that may not end up in the final product. A formal review process may waste time reviewing prototypes and quick solutions that may end up changing completely in the next iteration.

3.1.3 Test Cases

Both the Core and Exemplar teams wrote test cases to ensure that modifications to code would not break features. We used Jenkins, an open source continuous integration tool, to run our test cases every time a commit was pushed to the `master`, `dev`, `dev-core` or `dev-exemplar` branches on GitHub. This allowed us to verify that no commits to these branches introduced regressions, and if they did, quickly fix errors in the code that prevented test cases from passing.

3.1.4 Collaboration Tools

A major decision for the project was which version control system to utilize. We decided that Git would be the best system for our purposes. Its distributed nature allows offline development in a sane way. The lightweight branching and merging system would work well for isolating features and keeping both subgroups updated. A drawback of Git is that it may be difficult for software engineering students to learn compared to something like Apache Subversion (SVN). However, since SVN would likely be used with a single branch, it comes with its own set of problems, namely a high chance of merge conflicts during every day work. The online requirement may also prove annoying for students with unreliable

Internet access. We decided that the benefits of Git outweighed its higher learning curve, but ultimately there is nothing stopping students from copying the source code into a different version control system for their own projects.

Naturally, the version control system needs to be hosted somewhere. Our choices were comprised of the WPI-hosted FusionForge instance and a number of cloud-based solutions like GitHub and Bitbucket. GitHub was the clear winner because of the many points in its favor. Recall that Professor Pollice wants WPI Suite TNG to become a thriving open source project that other schools can use in their own software engineering classes. There is no better example of thriving open source communities on a free hosting provider than on GitHub. The site has become enormously popular and synonymous with Git itself. It has even become the home for the Linux kernel, several Mozilla projects, popular web frameworks, and millions of other repositories. The future viability of WPI Suite TNG relies on making contribution easy, and none of the other platforms can compare in this regard.

There are several other benefits to GitHub beyond pedigree. GitHub's tools for comparing branches, viewing history, and commenting per-line on commits are invaluable for the code review process. Forking the project is as easy as clicking a button, which takes away the administrative overhead of setting up a FusionForge project and SVN repository for each software engineering team. The Pull Request system will make it easy to pull in new modules from software engineering teams, and will also serve to formalize the code review process in the future. The public Issues system will allow the future community to report problems, request enhancements, and generally steer the project in the right direction. The Wiki feature makes it very easy to write and publish documentation. Finally, GitHub supports post-commit hooks that can update continuous integration systems or even social media accounts.

Perhaps the most important benefit of GitHub is that it will maximize the career impact for software engineering students who contribute to the project. Being involved in a public project on a popular site like GitHub is a great thing to include on a résumé. Potential employers can observe the WPI project experience first-hand. They can see code samples, how students work in a team, and be confident that students have version control experience. On top of this, GitHub Enterprise is becoming increasingly popular, so students' experience may even translate into direct experience with tools being used at a software company.

In order to keep our repository organized, we drafted a set of repository guidelines that outline how our branching system works, standards for commit quality, and some very basic style guidelines. Our branching model follows the *Git Flow* model proposed by Vincent Driessen (Driessen, 2010). The diagram in Figure 9 outlines the branching model and specific details on our adaptation can be found in the documentation included in the appendix of this paper. The basic idea is that you have two main branches: `master` and `dev`. The `master` branch always contains the latest stable, production code. The `dev` branch contains the most recent code that developers are working on, and may be unstable. Work is done on *feature branches* that split from the `dev` branch and are merged back in when finished. When the code on the `dev` branch is ready to be released, a *release branch* is split off that is eventually merged into `master` once stable. We slightly modified this model for our use – the exemplar module and core sub teams each have their own `dev` branches to avoid frequent conflicts, e.g. in the case of a core API change. Core changes were merged into the exemplar branch and tested, then merged into the main `dev` branch. Future development will abandon these team branches.

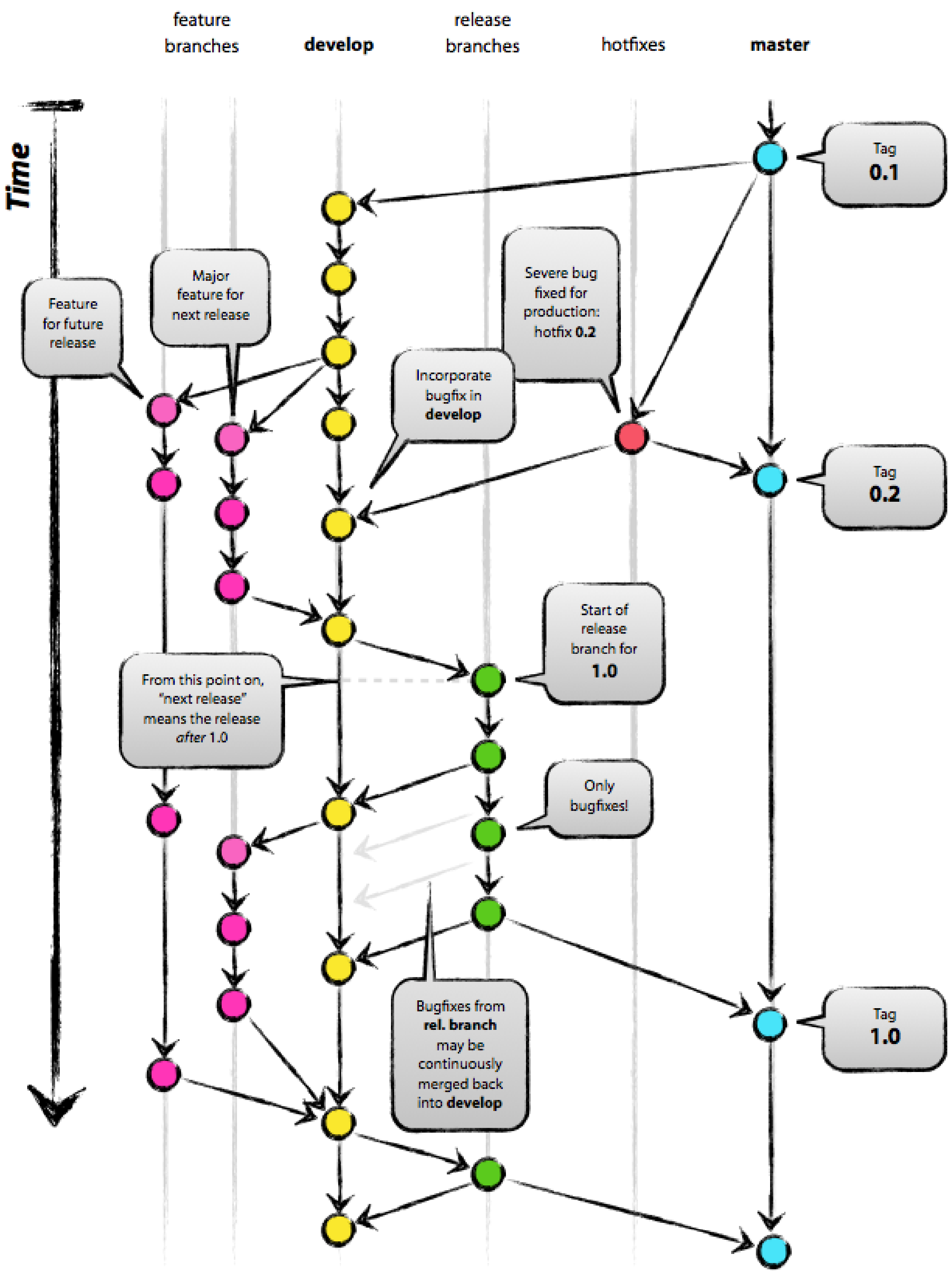

feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Figure 9: GitFlow Branching Model

The system for project documentation is another important consideration. We decided that most documentation should sit in the Javadoc comments in the code itself. Documentation that sits close to the code itself is more likely to be read and kept up to date. However, some documentation does not have a sensible place in the code – high level architecture documents, governance policy, development environment setup, etc. For this kind of documentation, we decided to use GitHub's built-in wiki. Wikis are useful because they can be viewed on the web, do not require much technical knowledge or overhead to edit, and allow easy linking and formatting.

An unfortunate issue with GitHub wikis is that the process for connecting documentation to particular commits or branches is not straightforward. For example, if one wanted to document a feature that has not been merged into the main branches, the only reasonable way to do so is to add to the existing global documentation. The wiki system is itself versioned with Git, but in a separate repository, so there are potential solutions involving adding the wiki to the main repository as a sub module. One could also hold static HTML documentation in the repository itself, but this adds maintenance overhead and makes reading the documentation more difficult. These solutions seem inelegant - solving this problem can be the subject of future work, or perhaps even a future feature of WPI Suite TNG.

Communication is a crucial part of any team project. The two sub teams mainly used email to communicate with each other. In order to avoid spam and better organize the project, two other channels of communication were also used. The first was PivotalTracker – a web-based project management system that supports tracking of the progress of features and bug fixes and discussion of implementation details. The tool also helps track the team's overall velocity and allowed the advisor to monitor team progress. The second communication channel was IRC through a *#wpisuite* channel on the freenode network. The exemplar module sub team generally used this to ask quick questions, perform code reviews, and hold informal meetings without the overhead of setting up a physical meeting. This channel will remain online for future contributors looking for a place to communicate, as is the common practice of most popular open source projects.

3.1.5 Pivotal Tracker

Pivotal Tracker is a planning tool for software projects with a focus on *stories*. A story, or user story, is a way of concisely defining a software requirement that is used frequently within the agile community. They are described in detail in Mike Cohn’s book, *User Stories Applied* (Cohn, 2004). The WPI Suite MQP used Pivotal Tracker for requirements and bug tracking. We created an epic for each general requirement. For example, one of our epics had the title, “Defect Tracker: Basic Issue Management”. This epic was created in order to represent the need for basic CRUD operations for defects as well as what basic fields each defect should support. CRUD is an acronym for create, read, update, and delete – the four main actions that can be performed on an entity.

Each epic has a list of stories that are linked to it. A story is a task which, when completed, adds a feature, fixes a bug, finishes a chore or creates a release. For example, the “Basic Issue Management” epic had a feature story titled, “Users need to be able to create a new issue”. Once completed, the team member responsible for the story marked it as *delivered*. At this point, another team member checked over the code and *accepted* the story. This acceptance allowed for the first team member to merge the code into the `dev-exemplar` branch. The second team member could choose to *reject* the story instead, indicating that it still needed work until returning to the *delivered* state for another review.

Each iteration, a team would assign various stories to its members. The Exemplar team accomplished this by consensus. Occasionally, a new feature would be introduced or a bug would be found in the middle of an iteration. When this occurred, a team member would create a story for it and would collaborate with the other team members to assign it. Generally, team members who had worked on a feature would address bugs regarding that feature. For new features, the team assigned members who were most familiar with the relevant components and tools.

3.2 Technologies Used

Most of the technologies used on this project are a result of the chief requirement: all code must be JVM-compatible. This requirement led to the core team choosing Java Servlets and db4o to develop the WPI Suite core. It also motivated the Exemplar Team to use Swing to implement the client application.

3.2.1 Swing

The exemplar module team decided to implement the desktop client in Swing. Swing was chosen because most students have at least some experience developing in Swing and it is one of the most commonly used GUI frameworks for Java. It is also relatively simple to modify and extend existing Swing GUIs, making Swing an ideal tool for an exemplar module that is intended to be easily understood and extensible.

The exemplar team also considered using a Web interface, but decided against it for a few reasons, chief of which was it would be too difficult for software engineering students. Using a web interface would potentially require students to learn 5 languages (Java for server-side logic, a templating language, HTML, CSS, and JavaScript). WPI offers only a single 4000-level Webware course, so it is unlikely that students would come in with experience in this area. Web development is a minefield of deprecated functionality, poor code examples and documentation, and browser quirks. If cruft spelled the end of the original WPI Suite in a standardized single language environment, novice students would surely do even worse in a web development environment.

3.2.2 REST API / JSON

The WPI Suite project decided to use a representational state transfer, or REST style API (Fielding R. T., 2000) with JavaScript Object Notation, otherwise known as JSON (Crockford, 2006) for communication between the client and server. This decision provides several advantages. First, it allows any client to interact with the core server. It makes it especially easy to implement a Web client in the future, perhaps as part of WPI's Webware class. Second, JSON is relatively easy for humans to read, which is important when debugging serialization issues. Third, it has ubiquitous library support. WPI Suite uses the Gson library to convert between native Java Objects and JSON representations of those objects (Google Inc., n.d.). The JSON representation makes it trivial to transmit model instances over HTTP requests between client and server.

3.2.3 Development Tools

We used several popular development tools while working on the project. The first of these is Eclipse (The Eclipse Foundation, 2013). It's the de facto standard IDE for Java development, and WPI tends to use it for many classes, so it made sense for us to develop with it and target it in our documentation. Eclipse's robust syntax highlighting, error detection, and warning system are vital for students who may be unfamiliar with Java. Being able to run unit tests and WPI Suite itself with the click of a button are important for speeding up the development process.

Another tool used was Ant (The Apache Software Foundation, 2013). Ant allows developers to specify a build process in XML format. This was necessary for building JARs and compiling code in an automated, consistent way. Building and testing the project is as simple as running one short command from the command line, even without Eclipse installed. We organized our build files such that there is one master build file in the root of the repository which recursively calls Ant on the build files within individual projects. Dependencies between projects are specified in another file in the root, and project build files can import a third file to inherit common functionality. This setup allows minimum redundancy between project build files without sacrificing the sovereignty of each project over its build process. Most new modules can simply copy a build file from an existing module and change some names to get it building properly. We considered more complicated systems like Maven, but decided that they were far too complicated and would take too long for us to understand and set up – not to mention our concerns over how well software engineering students could grasp the concepts (The Apache Software Foundation, 2013).

The final piece of the puzzle was Jenkins, a continuous integration system (About Jenkins CI, 2013). Each time a developer pushes a change to GitHub, GitHub notifies the Jenkins server. Jenkins then builds and tests WPI Suite TNG with the aforementioned Ant scripts. If the project fails to build or if there are test regressions, Jenkins emails the release engineer, and he can figure out who is at fault. This system was important in the later stages of the project where we needed to ensure stability before we unleashed the project onto students. Our Jenkins setup has some faults that would be nice to fix in the future. One problem is that it does not track breakage separately across branches. If a test is broken on branch A, and Jenkins then builds branch B where the test passes, then it considers the test *fixed*. Another problem is that there is no simple way to request a build of a specific branch or commit. This makes it difficult to publicly verify test results before merging feature branches into the `dev` branch.

3.3 Architecture Overview

WPI Suite consists of two main components: the Swing-based desktop client called Janeway, and the Java Servlet and db4o powered core. Communication between the client and core occurs using an HTTP REST API. Java objects are serialized into JSON in order to pass them in HTTP requests.

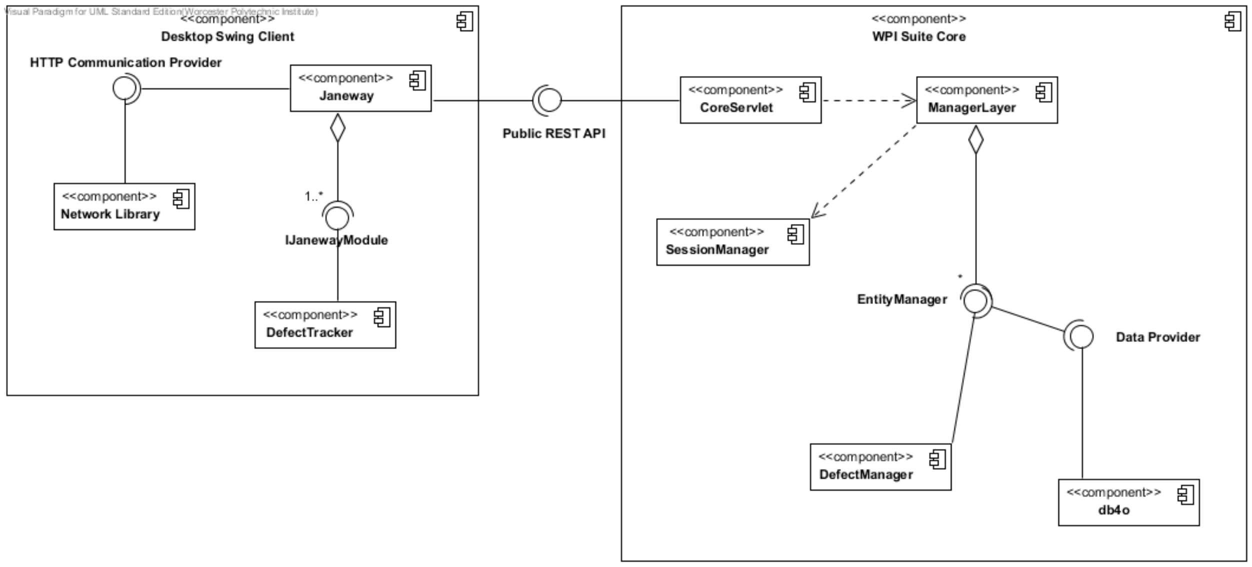


Figure 10: Architecture Diagram

The Desktop Swing Client component shown in the diagram above includes the Janeway application, a network library, and the exemplar module: Defect Tracker. Janeway is a Swing application that provides a basic framework for module user interfaces. The Network Library is a Java library developed by the project team to make forming HTTP requests for the core and handling responses relatively trivial for module developers. The Defect Tracker is a tool for tracking defects and the first module available for WPI Suite TNG.

3.4 Janeway

Rather than implementing a specialized Swing application that could display only the GUI of the Defect Tracker, the team decided to implement an extensible desktop client that can support multiple WPI Suite modules. Module developers can choose to write their GUIs using the Janeway module interface so that they can rely on some of the prebuilt GUI and convenience tools included with Janeway, making it possible to develop a working module GUI relatively quickly (as proven by the D term Software Engineering students). In addition, using Janeway makes it simple for the developer to make use of the network library that the team wrote to simplify the process of sending HTTP requests, receiving HTTP responses, and communicating with the WPI Suite core in general. The Janeway client can run on any system that supports Java - it has been tested on Windows, Ubuntu, and OS X.

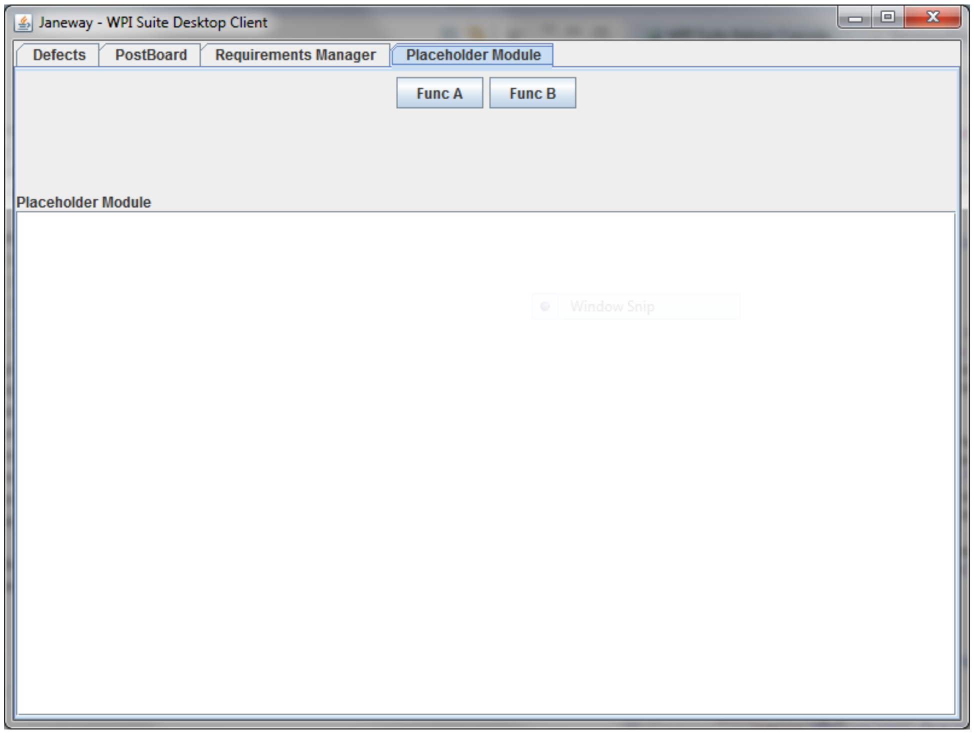


Figure 11: Janeway GUI

The user interface for Janeway is relatively simple. It consists of a frame that modules can embed their interfaces in. The frame is slightly more advanced in that it contains a tab bar that allows users to switch between modules. It also includes a toolbar area for each tab by default. In Figure 11 a placeholder module is being displayed in Janeway. The two buttons are in the toolbar panel for the Placeholder Module tab and the large white text field is in the main content area.

3.4.1 Architecture

The Janeway client provides five main services. The first is a set of interfaces that are implemented by modules so that they can include their GUIs in the Janeway client. The second is a module loader that allows modules to be dynamically loaded at runtime, eliminating the need for module developers to modify Janeway code to include their module. The third is authentication and project selection. Janeway handles login and project selection immediately when it starts up, and this information is later accessible to modules. Fourth is the ability for modules to register keyboard shortcuts with Janeway. This allows registered keyboard shortcuts to change based on which module tab in Janeway is selected and prevents keyboard shortcuts for inactive module tabs from being activated. Finally, a special singleton class called `ConfigManager` is provided that allows module developers to persist basic client configuration information locally on the client's machine.

Interfaces

The Janeway client provides an interface to module developers called `IJanewayModule`. The interface requires modules to implement two methods, one that returns the name of the module, and another that returns a list of the tabs the module will use (since modules can display multiple tabs if they desire). A class called `JanewayTabModel` is provided to represent the tabs. Each module's `getTabs()` method is expected to return a list of `JanewayTabModel`. This model class is where the module developer provides their GUI panels (of type `JComponent`). They are expected to provide a toolbar panel and a main panel. Optionally, they can provide a list of keyboard shortcuts to be associated with the tab.

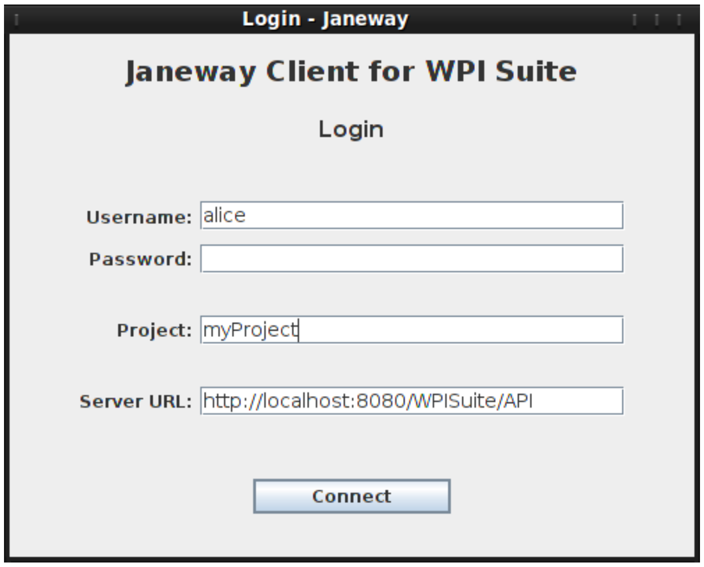
Module Loader

To prevent module developers from needing to modify Janeway code to load their module, a dynamic module loader was written and included with Janeway. The loader first looks at a `modules.conf` file located in the same directory as the Janeway client to determine in which directories it should look for module JAR files. Next, each JAR file in the module directories is opened and searched for a `manifest.txt` file. This manifest file indicates the name of the class that implements `IJanewayModule`. Each `IJanewayModule` that is encountered is then instantiated using reflection and loaded into the GUI.

Authentication

All WPI Suite clients must authenticate and log in before being able to interact with server side modules such as the Defect Tracker. This is accomplished by making a POST request to the login URL on the WPI Suite server. This post request must contain a header for basic authentication. If the user's credentials are correct, WPI Suite will return a session cookie.

In order to work with a project, a request must be sent to the server to select a project. This is accomplished by sending a PUT request to the login URL of the server. The body of this request must be the name of the project to select. If the project is successfully selected, the server will set a cookie in its response.



Username: alice
Password:
Project: myProject
Server URL: http://localhost:8080/WPISuite/API

Figure 12: Janeway Login Window

Janeway performs both these steps on its login window (pictured in Figure 12). The user is prompted to enter a username, password, server URL, and a project name. When the user is ready to log in, Janeway makes a login request to the server. Upon success, Janeway will then attempt to select the given project. If the project select request is successful, Janeway will close the login window and open the main window. If either the login request or the project selection request fails, the user will be notified via a dialogue box and relevant error message.

Keyboard Shortcuts

In order to support a different set of keyboard shortcuts for each module tab, Janeway registers its own `KeyEventDispatcher` to listen for key events. Each of the `KeyboardShortcut` objects provided by a module contains a `KeyStroke` and an `Action`. If the key event matches the `KeyStroke` in a `KeyboardShortcut`, the associated `Action`'s `actionPerformed()` method is called to execute the shortcut.

Configuration Manager

The last major component of Janeway is the configuration manager. The configuration manager allows developers to save basic information on the user's computer in a data file. By default, the configuration manager is used to persist the user's login name, project name, and server URL in the login form. However, developers can add fields to the `Configuration` class that store any type of information. Each time the Janeway client is closed the configuration is automatically written to disk.

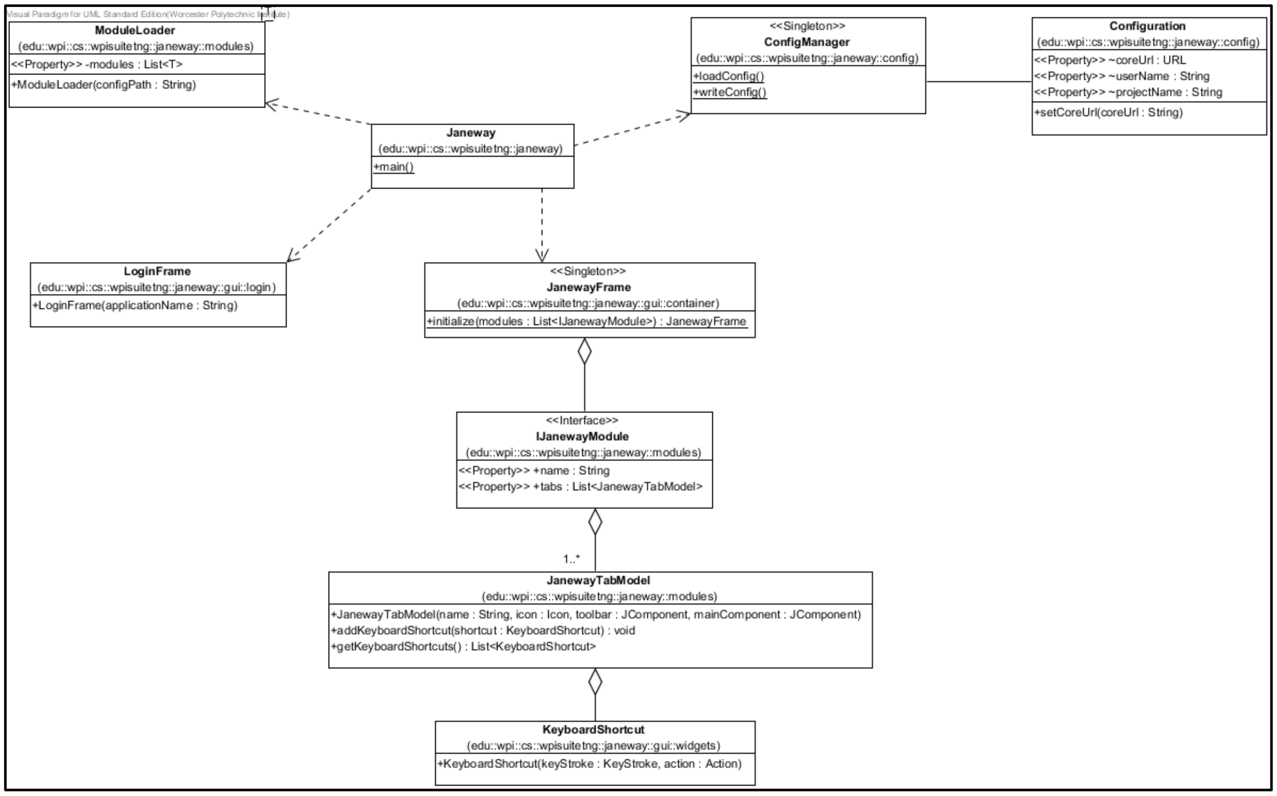

Visual Paradigm for UML Standard Edition(Worcester Polytechnic Institute)
ModuleLoader
(edu::wpi::cs::wpisuitetng::janeway::modules)
<<Property>> -modules : List<T>
+ModuleLoader(configPath : String)
<<Singleton>>
ConfigManager
(edu::wpi::cs::wpisuitetng::janeway::config)
+loadConfig()
+writeConfig()
Configuration
(edu::wpi::cs::wpisuitetng::janeway::config)
<<Property>> ~coreUrl : URL
<<Property>> ~userName : String
<<Property>> ~projectName : String
+setCoreUrl(coreUrl : String)
Janeway
(edu::wpi::cs::wpisuitetng::janeway)
+main()
LoginFrame
(edu::wpi::cs::wpisuitetng::janeway::gui::login)
+LoginFrame(applicationName : String)
<<Singleton>>
JanewayFrame
(edu::wpi::cs::wpisuitetng::janeway::gui::container)
+initialize(modules : List<IJanewayModule>) : JanewayFrame
<<Interface>>
IJanewayModule
(edu::wpi::cs::wpisuitetng::janeway::modules)
<<Property>> +name : String
<<Property>> +tabs : List<JanewayTabModel>
1..*
JanewayTabModel
(edu::wpi::cs::wpisuitetng::janeway::modules)
+JanewayTabModel(name : String, icon : Icon, toolbar : JComponent, mainComponent : JComponent)
+addKeyboardShortcut(shortcut : KeyboardShortcut) : void
+getKeyboardShortcuts() : List<KeyboardShortcut>
KeyboardShortcut
(edu::wpi::cs::wpisuitetng::janeway::gui::widgets)
+KeyboardShortcut(keyStroke : KeyStroke, action : Action)

Figure 13: Janeway Class Diagram

The diagram in Figure 13 shows the primary classes included in the Janeway client. The Janeway class contains the main method that starts up Janeway. The Janeway class uses the `ModuleLoader` to dynamically load modules and the `ConfigManager` to load data that was previously persisted on the client. The `JanewayFrame` class extends `JPanel` and is the main GUI view for the client. This view contains all of the `IJanewayModule` objects that contain the module tabs.

3.4.2 GUI Design Decisions

While developing the Janeway client, we made several high-level decisions about the user experience and GUI design. It is important to note that good interface design was not a high priority in this project – it was much more important to have a basic working product that is easy for students to extend. Consequently, some decisions may disregard good design standards for the sake of simplicity.

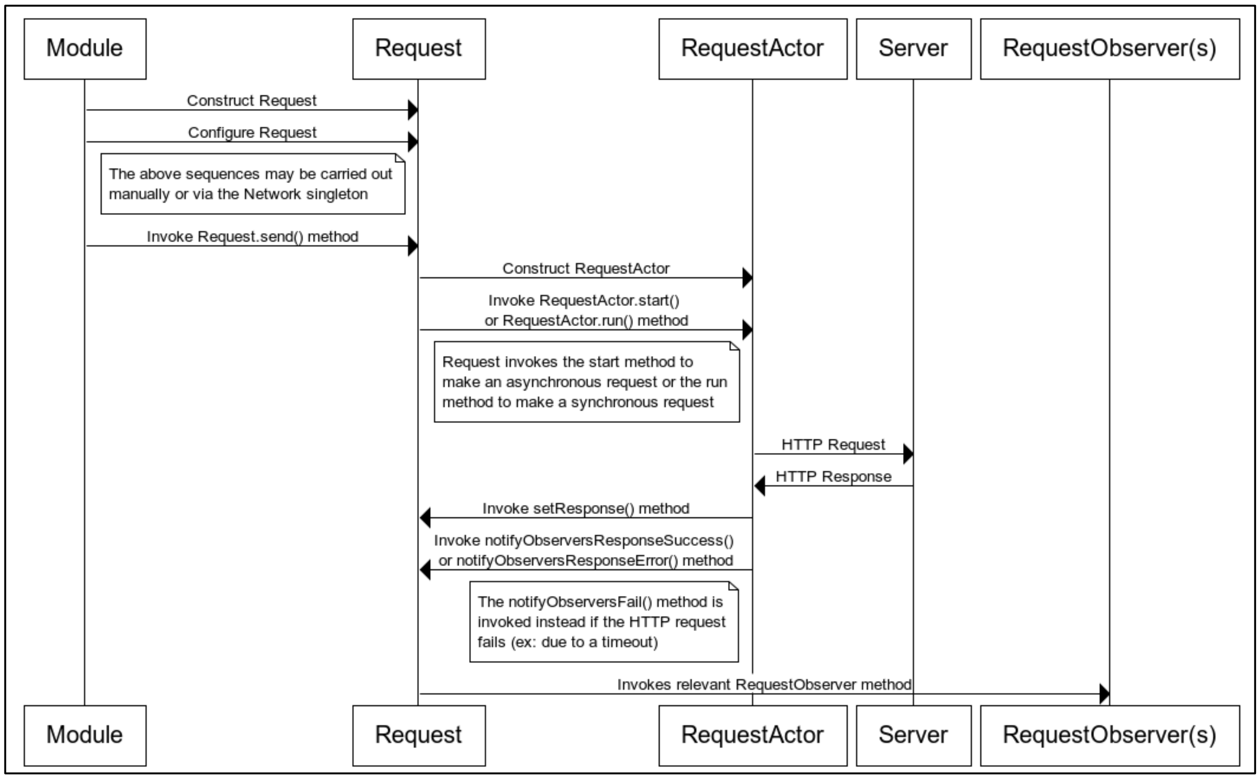
One decision we made early on was regarding the Swing *look-and-feel* (L&F). Swing has a set of standard components, but they can be styled differently depending on the chosen look and feel. By default, Swing will use a look and feel appropriate for the platform it is running on to make Swing components fit in with the operating system interface. Instead of this, we decided to always use the cross-platform *Metal* L&F regardless of operating system. This is certainly less visually appealing than operating system-specific L&Fs, but we thought it would not be a good use of our time (and students' time) to debug UI problems that only occur on certain L&Fs or to try following different design standards per-OS. This time sink became obvious very early on when we discovered crucial limitations with tabs on the *Aqua* L&F on OSX. Using native L&Fs may be the subject of future work.

Another design decision we made while creating Janeway was to implement a toolbar that modules can use. This toolbar contains named groups of UI elements (buttons, text inputs) separated with a large margin. We were aiming to use vertical lines to separate groups more obviously, but did not have time to implement that. The contents of the toolbar change depending on context – for example, when viewing a Defect it might contain a "Save Defect" button. The toolbar may not be the best UI for any given module tab, but we felt it was worth having so that there is some forced consistency between tabs and an easy place for students to start from.

A *widgets* package is included in the Janeway project that supplies useful Swing components that can be used by module developers. Currently there are `JPlaceholderTextField`, `Hoverable`, and `KeyboardShortcut` classes in the package. The text field is a simple subclass of the built-in `JTextField` that displays grey text by default that is then removed when the user clicks on the field. `Hoverable` is an interface that requires implementing classes to respond to hover events. Lastly, the `KeyboardShortcut` class is used by modules when they want to register keyboard shortcuts with Janeway. A `KeyboardShortcut` contains two main fields: a `KeyStroke` indicating the key combination and an `Action` to be performed when the shortcut is typed.

3.5 Network Library

The network library exists to simplify performing HTTP requests for Janeway module developers. It exists as a separate library to allow for reuse, since it is not dependent on Janeway in any manner. There are six primary components of the network library: the `Network` singleton, the `NetworkConfiguration` class, the `Request` class, the `RequestActor` class, the `RequestObserver` interface and the `ResponseModel` class.



Module
Request
RequestActor
Server
RequestObserver(s)
Construct Request
Configure Request
The above sequences may be carried out manually or via the Network singleton
Invoke Request.send() method
Construct RequestActor
Invoke RequestActor.start() or RequestActor.run() method
Request invokes the start method to make an asynchronous request or the run method to make a synchronous request
HTTP Request
HTTP Response
Invoke setResponse() method
Invoke notifyObserversResponseSuccess() or notifyObserversResponseError() method
The notifyObserversFail() method is invoked instead if the HTTP request fails (ex: due to a timeout)
Invokes relevant RequestObserver method

Figure 14: Network Request Sequence Diagram

The `Network` singleton allows any module developer to easily create preconfigured requests. `Network` creates these requests using an internal default `NetworkConfiguration`. `NetworkConfiguration` instances store the API URL for the WPI Suite server, a set of default request headers, and a default set of `RequestObservers`. An example of a default header would be the session cookie from logging into the server. The default set of `RequestObservers` can be used if, for instance, Janeway developers wanted to make a log of every response that Janeway received via the Network library. The `Network` singleton's default `NetworkConfiguration` is not guaranteed to be used in every `Request`.

The `Request` class follows the observable pattern and represents an HTTP request. It also allows the user to send an HTTP request via the send method. It may be constructed via the `Network` singleton, in which case it will start with the same attributes as the default `NetworkConfiguration` or it may be manually constructed. A `Request` might be manually constructed if a module developer wants to make a request to a server other than WPI Suite. A `Request` also manages read and connect timeout lengths. `Requests` can be either synchronous or asynchronous. By default `Requests` are asynchronous, but they may be set to synchronous if a developer desires. This is especially useful in testing, where the developer may want the `Request` to block after being sent. When the send method is called, the `Request` creates a new `RequestActor` and either calls its start method, if asynchronous, or its run method, if synchronous.

Using asynchronous requests by default is an important design decision in itself. If a developer used a synchronous request, it would block the main thread until the client received a response. Some developers would certainly be tempted to do this, since a synchronous request is much easier to code. Unfortunately, this has the very visible side effect of blocking all UI interaction, which results in a horrible user experience. Imagine loading a page in a web browser only to have the interface freeze up, then waiting for the entire page to load until your browser unfreezes. We decided to reduce the chance of problems like this by making requests asynchronous by default.

The `RequestActor` is where the request is made and the response or errors handled. It extends `Thread` so that it may make a non-blocking request if needed. The `java.net` library is used to make an HTTP request to the server. After making the HTTP request, it will construct a `ResponseModel` and set it as the parent `Request`'s response. It will then call either the `notifyObserversResponseSuccess`, `notifyObserversResponseError` or `notifyObserversFail` method of the parent `Request`. These methods will call the relevant method of every observer of the parent `Request`.

The `RequestObserver` interface is based on the observer pattern and has three methods that must be implemented: `responseSuccess`, `responseError` and `fail`. These three methods are meant to be called by a `Request` upon completion or failure. The `responseSuccess` method is called when a request successfully completes and receives a response with status code from 200 to 299 (inclusive). It is provided with an `IRequest` interface which contains methods for retrieving information about the HTTP request as well as the response to that request. The `responseError` method is called when the response has a status code outside the 200 range. This signifies that there was an error returned by the server. The `fail` method is called when a request is unable to be completed. This may be due to a timeout or other issue that would prevent proper communication with the server. Its parameters include an `IRequest` interface and an `Exception`. This `Exception` is the one thrown by the `java.net` library.

3.6 Defect Tracker

The Defect Tracker module provides simple defect tracking features. The main goal of the Defect Tracker module is to serve as an example of how to develop a WPI Suite module. It uses most of the features provided by the WPI Suite core and all of the Network library functionality. The defect tracking features that have been implemented include defect creation, editing, tagging, assigning, commenting, change tracking, and browsing.

3.6.1 Core Interaction

The basic functionality of the core is to provide data persistence for modules. This is accomplished by sending and receiving `Model` objects. The core provides an interface called `Model` that is implemented by model classes. For the Defect Tracker module, the primary model is called `Defect`. This class is passed back and forth between the Janeway client and the core in serialized form. The Defect Tracker module also provides a class that implements the `EntityManger` interface. This class exists in the core and is used by modules to provide module-specific logic for creating, retrieving, updating, and deleting models.

Models

`Model` classes primarily contain a number of fields that store data about the model. For example, the `Defect` model has a number of fields including assignee, creator, description, and title. Models only need to implement the `Model` interface when they will be directly sent and received from the core. Every `Model` must have an associated `EntityManager` residing in the core.

To make JSON serialization and deserialization more transparent, a `toJSON` method is required of all classes that implement the core's `Model` interface. The `toJSON` method is expected to return a JSON string representing the object. In practice, most of the `toJSON` methods are implemented in the same way; they delegate serialization to Gson (a library provided by Google). A `fromJSON` method is not required, however it is recommended that all models implement a static method called `fromJSON` that takes as an argument one `String` containing a serialized object of the same type as the model and returns a new instance of the model. By implementing both `toJSON` and `fromJSON`, all of the core interaction classes, including the entity managers, can simply use these methods instead of needing to construct and call their own Gson objects.

Entity Managers

In order to implement the Defect Tracker module, the exemplar team had to create implementations of necessary core interfaces. The most important of these is the `EntityManager`. `EntityManagers` handle certain network requests that the server receives. Each `EntityManager` needs to register itself for a URL based on the entity it manages, such as /API/defecttracker/defect. The core dispatches requests to the appropriate method based on the HTTP method used by the request – PUT requests are sent to `EntityManager.makeEntity()`, POST requests are sent to `EntityManager.update()`, etc. The return values of these methods determine the response to the client, and the methods can throw exceptions that translate into responses with appropriate HTTP error codes.

We decided to split up the functionality within `EntityManager`s. One common responsibility is validation, since updating the information in a `Defect` model requires you to make sure that it has a title, the author is an existing user, and so on. We followed a pattern of creating `ModelValidator` classes that handled this responsibility for each model and returned a list of `ValidationIssues`. This makes the `EntityManager` methods much cleaner – they simply call `ModelValidator.validate()` and throw an exception if there are any issues. Moving this responsibility into a separate class will also allow future reuse client-side – clients can mock out any necessary parameters like the database API.

To understand the next responsibility, it is necessary to explain the *disconnected object problem* present when using db4o. As an object database, saving to db4o is as simple as passing the Java object itself to a `save` method. To avoid saving duplicate objects instead of updating existing ones, db4o keeps track of the identity of objects you retrieve from the database. If you save an object that resides in the same memory location as an object that was retrieved earlier, db4o will update that existing object's fields. The problem is that this memory relationship is lost when sending objects over the network. If you parse out a JSON object from the network in a request that is supposed to update an existing object, you will end up with a duplicate object if you directly save the parsed object. Consequently, the only way to process an update is to pull the original object out of the database, copy fields from the parsed object to the original object, and then save the original object.

Another class named `ModelMapper` handles the copying responsibility. It uses reflection to find all getter/setter pairs in a model and copy all fields from one model to another. The alternative is to manually copy each field, but this can be cumbersome if a model has many fields. The developer would also have to write such a method for every model. Another problem with the manual approach is that if you forget to update your copying function after adding a field to a model, you will have a subtle bug where that field does not update properly.

Another requirement makes manual copying untenable: Defects need to keep a history of all changes. To do this manually, you would need to add another few lines of code to *every* field that is being copied during an update. With the `ModelMapper`, we simply added a callback function that allows us to track each field change being made. This may seem overcomplicated, but it has served us well and the software engineering teams have reused this functionality.

Client Communication

Using the built-in Network library included with Janeway makes client communication with the core relatively simple. Janeway and Defect Tracker both roughly follow the model-view-controller pattern, so the controllers are where requests are sent and responses are handled. The basic pattern is that the controller constructs a `Request`, adds a `RequestObserver` to the request, and then sends the `Request`. Since the Network library handles threading automatically, the controller does not block to wait for a response. Instead, the `RequestObserver` is notified when the response is received from the core. Depending on how the `RequestObserver` is implemented, it can either directly process the response received and update the view, or it can call a method in the controller and pass the response data back to the controller so that the controller can decide how to proceed with the response. It is worth noting that there are three methods in each `RequestObserver` to handle response success, failure, and error respectively. This prevents module developers from needing to check the response code directly.

3.6.2 GUI Design

In order to demonstrate the structure of a working WPI Suite TNG module, we had to develop one. We chose to create a defect tracker since it is a fundamental project management tool that is relatively straightforward to implement. Since future students will be using this defect tracker to track bugs in their modules, we wanted to make one that would be functional for those users. As discussed in the background of this report, we examined existing defect trackers to determine a feature set and user experience that would fit our needs.

It was also important to examine the failings of the defect tracker in the old WPI Suite, which some of the team used when they took Professor Pollice's software engineering class themselves. The original defect tracker opened up to show a table of all defects. To edit or view a defect, one would click on a defect in the table, and an editor would pop up on the bottom of the screen. This approach is fundamentally flawed – there is no way to view one defect while editing another (perhaps you needed to look something up in a different defect, or your work in one defect was interrupted to work on a different one).

To allow multitasking, a tabbed interface was implemented. Users can open multiple defect tabs at the same time to compare or reference them. They can even open multiple lists of defects – perhaps one list contains all defects assigned to them, while another contains un-triaged defects. The tabbed interface additionally achieves parity with web browser interfaces, which typically use tabs to keep track of multiple web pages. This is important because our advisor has plans to extend WPI Suite TNG for use in his Webware class, so the defect tracker interface would have to be replicated on a web site.

The original WPI Suite defect tracker was lacking a couple of other features that made it impractical to use in a real project environment. It had no way to add comments to a defect, and this feature was sorely missed. There was no good way to communicate about the status of a defect, ask for clarification from the original reporter, etc. Students had to resort to holding conversations by editing the free-form description text field. Another problem was that defects did not track their own change history. If someone resolved a defect for example, there was no accountability, and no way to talk to the person responsible besides out-of-band communications. We knew that we had to include comments and history in order for our defect tracker to be a viable product.

A major user experience decision was how editing should be handled. This feature varied among the existing defect trackers we researched in the background of this paper. Some of them had a distinct *edit mode* – fields could not be changed until the user clicked a button. We disliked this approach because it adds another click to a very common operation and introduces more complexity (to both the user and code) in the form of modes. The other approach was to leave fields always editable, but this has a problem of its own. Most implementations do not indicate that you have changed a field from its original value. If the user is adding to a defect's description and then is interrupted, when they return they must remember that they made changes. To solve this problem, we highlight changed fields to remove this additional user memory requirement. This fix made the modeless editing technique the clear winner.

3.6.3 Features

Basic Defect Creation and Editing

The Defect Tracker module for Janeway provides a means for creating and editing defects via a single interface. This interface provides all the available fields for defects and allows the user to edit them. After editing, the user presses the Save Defect button to create or save changes to the defect. The module will determine whether or not it should make a request to create a new defect or update an existing one and then make the request using the Network library. While the request to the server is made, the fields are disabled to prevent the user from making changes until a response is received. After receiving a response, the fields will be enabled. If the request fails, the user will be notified via a message box and the fields will be enabled.

Defect Fields

Every defect has a title, description, status, created date, modified date, creator and assignee. In order to create a defect, the title and creator fields must be present. When the user attempts to save or create a defect, the fields are validated and if there are violations, the user is notified and the new field value is not saved.

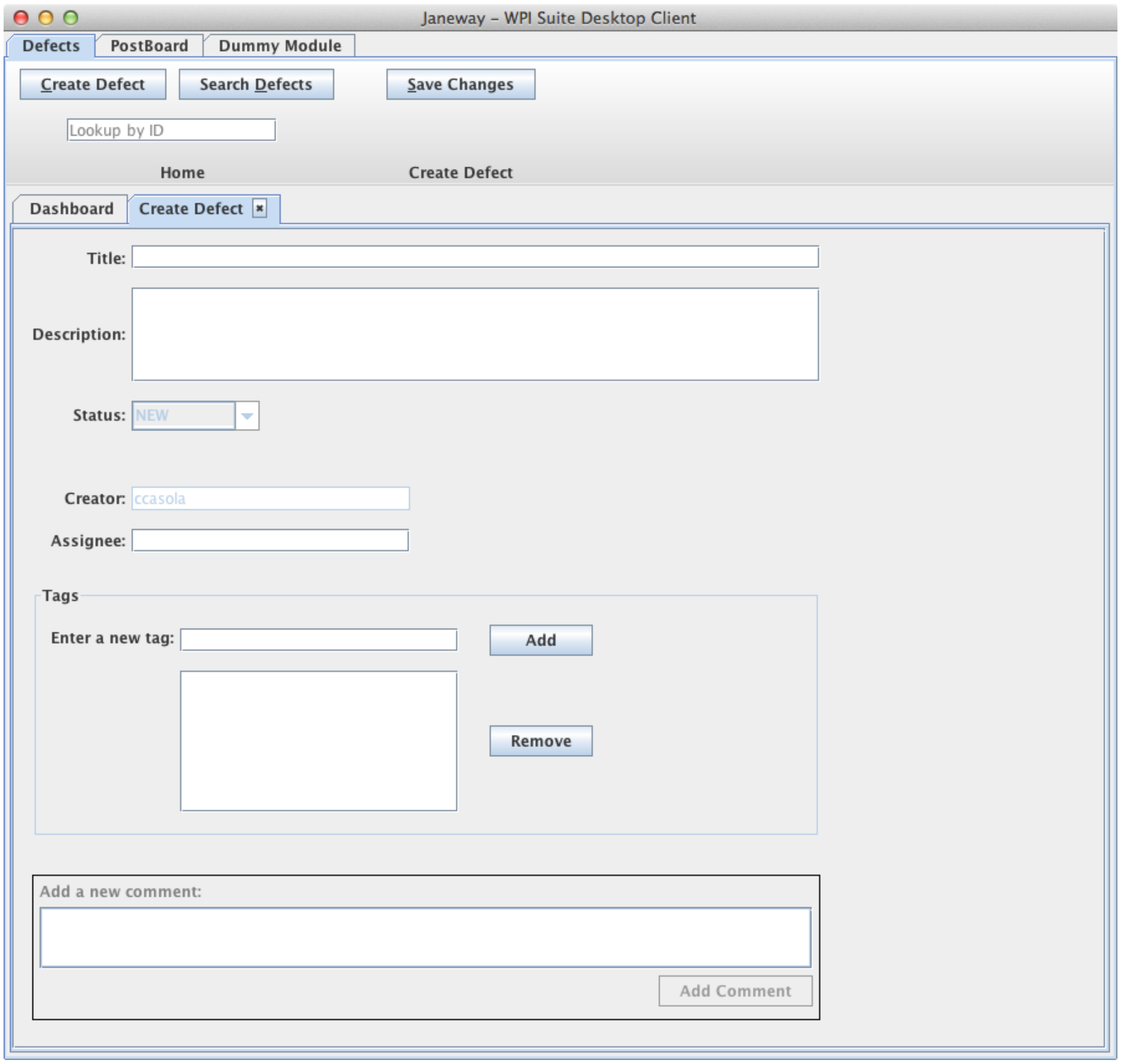
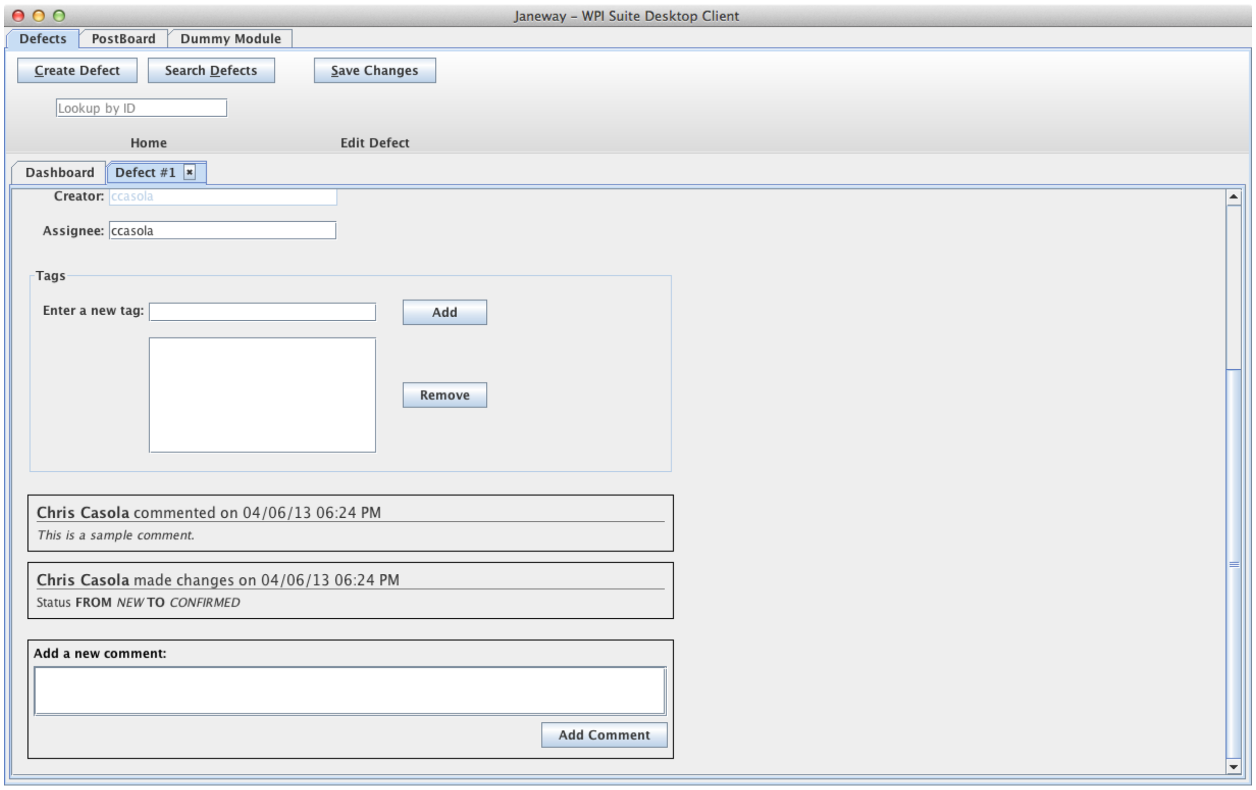


Figure 15: Defect Form

The figure above shows the defect creation and editing form. In the screen shot commenting is disabled because this is a new defect that has not yet been saved. Once the user fills out the title and description and clicks the *Save Changes* button, the defect is saved to the core and comments are enabled.

3.6.4 Comments and Change Tracking

Users have the ability to add comments to a defect. Comments are displayed in sequential order beneath the fields of a defect. Each comment displays the name of the person who commented and the date and time the comment was made. In addition to comments, changes to the defect's fields are also recorded and displayed. These changes are interleaved in the list of comments that are displayed below the defect fields.



Creator: ccasola

Assignee: ccasola

Enter a new tag:

Chris Casola commented on 04/06/13 06:24 PM
This is a sample comment.

Chris Casola made changes on 04/06/13 06:24 PM
Status **FROM** *NEW* **TO** *CONFIRMED*

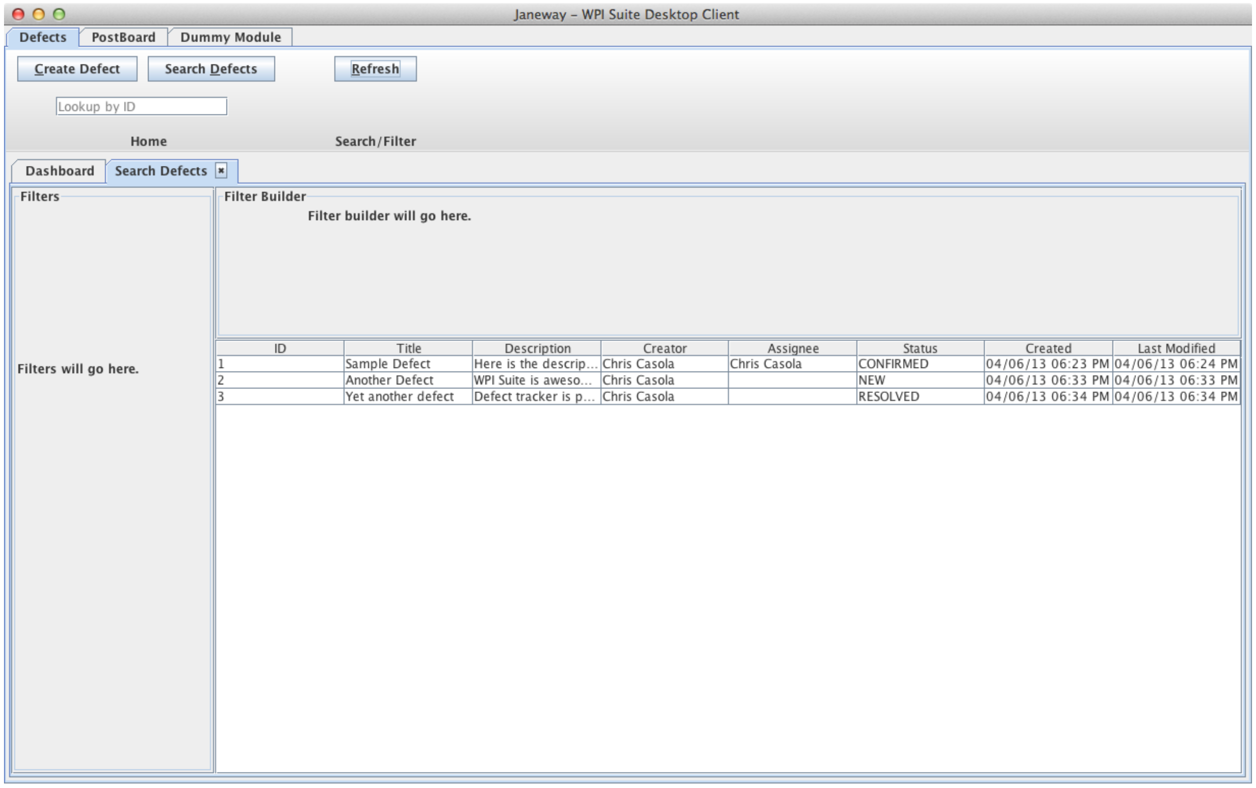
Add a new comment:

Figure 16: Defect Comments and Changes

The figure above shows that the user named "Chris Casola" commented on the defect and then proceeded to change the status from *NEW* to *CONFIRMED*. The change and comment log is updated as the user makes changes to the fields or writes additional comments.

3.6.5 Browse and Lookup Defects

Once a defect has been entered into the system other users will want to be able to view it. This is done using the *Search Defects* button located in the Defect Tracker toolbar. When the button is clicked, all defects are retrieved from the server and displayed in a table on a new tab.



ID	Title	Description	Creator	Assignee	Status	Created	Last Modified
1	Sample Defect	Here is the descrip...	Chris Casola	Chris Casola	CONFIRMED	04/06/13 06:23 PM	04/06/13 06:24 PM
2	Another Defect	WPI Suite is aweso...	Chris Casola		NEW	04/06/13 06:33 PM	04/06/13 06:33 PM
3	Yet another defect	Defect tracker is p...	Chris Casola		RESOLVED	04/06/13 06:34 PM	04/06/13 06:34 PM

Figure 17: Browse Defects

As shown in Figure 17 above, all of the defects in the system are displayed in a table. The table can be sorted by any of the columns. Double-clicking on a row in the table will open the corresponding defect in a new tab for viewing and editing.

If the user knows the id of a specific defect they would like to view or edit, they can enter that ID in the Lookup field on the Defect Tracker toolbar. Typing an ID followed by the Enter key will open a new tab with the requested defect. It is possible to have multiple defect tabs open enabling the user to edit and view multiple defects simply by switching between tabs. For convenience, they can also switch tabs using a keyboard shortcut (CTRL + TAB).

3.7 Issues Encountered

This section describes some of the major issues that were encountered during the development of the Defect Tracker and Janeway client. The steps that were taken to resolve the issue are also provided. Future module developers will likely encounter similar issues to those outlined here and can make use of the solutions provided.

3.7.1 Swing Layout Managers

Developing the user interface for the Defect Tracker involved using a number of built-in Swing classes including layout managers. Swing's layout managers are used to control the position of GUI widgets and control how they reposition themselves as the size of the window and the size of the widgets change. The particular layout manager the developer chooses has a large impact on how much control the developer has over positioning. Generally, the more control the developer has over layout, the more complex the layout manager is to use.

The defect form shown in Figure 15 uses the `SpringLayout` manager. This manager was used because it is one of the most powerful and flexible. However, the effort required to layout the form exactly how we wanted was very high. There were also a number of bugs in the interface caused by the complexity of `SpringLayout` that were difficult to find and resolve. We would recommend that inexperienced module developers start by using the simpler layout managers like `FlowLayout` and `GridLayout`. By using simpler layout managers and nesting panels within panels, sophisticated layouts can still be achieved.

3.7.2 Deserialization

There were two situations encountered during the development of Defect Tracker where the Gson library was unable to deserialize an object without some manual effort. The first is when trying to deserialize an object that contains a field with an unknown generic type parameter. The `DefectChangeset` class is an example, it has a field called changes that is of type `Map<String, FieldChange<?>>`. The unknown type of `FieldChange` is where the issue arises. To overcome the problem, a custom deserializer was written called `DefectChangesetDeserializer` (see Figure 18). To enable Gson to use the custom deserializer, a static method is added to `Model` classes that rely on the deserializer called `addGsonDependencies`. This method constructs the custom deserializer and adds it to the `GsonBuilder` that is passed as the single argument. The model's `fromJSON` method can then call the static `addGsonDependencies` method before deserializing to ensure the necessary deserializers are available.

```
public class DefectChangesetDeserializer implements
JsonDeserializer<DefectChangeset> {

     @Override
     public DefectChangeset deserialize(JsonElement json, Type type,
     JsonDeserializationContext context) throws JsonParseException {
```

Figure 18: Custom GSON Deserializer

A custom deserializer is also required for fields of an abstract type. The reason is that when deserializing, the Gson library has no way of knowing which concrete type to instantiate. By providing a custom deserializer, the content of the JSON being received can be manually analyzed to determine the concrete type of the object before deserializing.

3.7.3 Request Observers

Early versions of the Network library relied on Java's built in `Observer` interface to represent request observers. The `Observer` interface has an `update` method that takes an `Observable` and an `Object`. The issue arose because a `Request` (which implemented `Observable`) was always the type being passed to the `update` method, leading to module developers needing to cast from `Observable` to `Request` every time the `update` method was called. To avoid this, the Network library now includes a custom `RequestObserver` interface with three methods: `responseSuccess`, `responseError`, and `responseFailure`. The new interface solves the casting problem and simplifies error detection. Module developers no longer need to explicitly check the HTTP response codes because the Network library performs the check for them and simply calls `responseSuccess` if the code is a 200, `responseError` for all other HTTP response codes, and `responseFailure` if the request cannot be sent or the request times out.

3.7.4 Keyboard Shortcuts

While developing Janeway and the Defect Tracker GUI, it became clear that allowing modules to register their keyboard shortcuts in the standard way would cause a number of issues. If two modules registered the same shortcut, users could unintentionally activate two shortcuts, resulting in confusion. Since Janeway displays modules in their own tabs and only one tab can be active at a time, it also makes sense that only keyboard shortcuts for the active tab should be enabled. This is not enforceable if the built-in keyboard shortcut mechanism is used. To solve the problem, a custom `KeyEventDispatcher` is included in the Janeway implementation. Modules can provide keyboard shortcuts for each of their Janeway tabs by returning a list of `KeyboardShortcut` objects from the `getKeyboardShortcuts()` method in their `JanewayTabModel` objects.

The `KeyboardShortcut` class was written by the project team to make use of some of the built-in shortcut functionality. The `KeyboardShortcut` class contains a `KeyStroke` and an `Action` to represent the key combination and the action to perform. When the user presses a key combination, the Janeway `KeyEventDispatcher` checks through the list of keyboard shortcuts for the currently active tab and activates the action of shortcuts with matching `KeyStrokes`. In order to support global keyboard shortcuts (e.g. a shortcut for switching module tabs) there is a special purpose method named `addGlobalKeyboardShortcut` in Janeway's `KeyEventDispatcher`.

3.7.5 Reliance on the Core

During early development of Janeway and Defect Tracker, the team faced the difficulty of developing against an API that was not yet implemented or clearly specified. To manage this, the team focused on developing the Network library and the Janeway desktop client (including dynamic module loading) first. A `MockServer` was also created to mimic the anticipated behavior of the WPI Suite core.

Once a basic core was implemented, both teams still had to develop features before the other team supported the feature. The exemplar team overcame this by relying on the `MockServer` and other unit testing strategies. `MockRequest` and `MockNetwork` classes were created to help test client code that sends requests to the server by intercepting those requests and faking the responses. The `MockNetwork` class allows testers to retrieve the last request sent and then inject a fake response.

The stability of both the core and the client during development was not always certain. Both teams benefited from unit testing and clear definition of the interfaces and API to deal with instability. An added benefit of the situation was each team had a whole other team to quickly detect and report bugs.

3.7.6 Repository Guidelines

Early on in the project repository guidelines were setup to outline rules and best practices for committing, branching, and merging within the WPI Suite project's Git repository. Issues arose when mistakes were made because some merges and commits broke stable branches requiring someone to go in and re-stabilize those branches. Stricter adherence to the GitFlow methodology as specified in the repository guidelines would have helped to mitigate this issue (Driessen, 2010).

4 Results & Analysis

The two main goals for this project were to (1) provide a challenging but achievable project for software engineering students and (2) to address the lack of tools available for managing software engineering student teams. The project's success can be measured based on how well these goals are satisfied. Since the project team was able to pilot the new version of WPI Suite on a software engineering class at WPI, there was an opportunity to observe how well the student project teams responded to WPI Suite and to collect feedback in the form of a survey. Other more traditional methods of software evaluation, including code metrics and test statistics can also be used. Finally, some known issues and ideas for future improvement are discussed.

4.1 Surveys

A survey was conducted of the D term CS3733 students to determine the efficacy of WPI Suite as a learning tool for software engineering students. The sample size was 38 students out of the 73 students initially registered for the course. The survey was administered online via the course webpages and responses were completely anonymous and optional.

4.1.1 Demographics & Prior Knowledge

The first set of questions aimed to determine some demographic information about the students as well as the level of their relevant technical skills before beginning the course. Based on the responses, 68% of the students were sophomores and 29% were juniors, with one student choosing not to answer. The class was composed of students in a wide range of majors including Computer Science (71%), Robotics Engineering (24%), IMGD (11%), and ECE (5%). Another 15% of students were not majoring in a computer-related field. This data is shown in Figure 19.

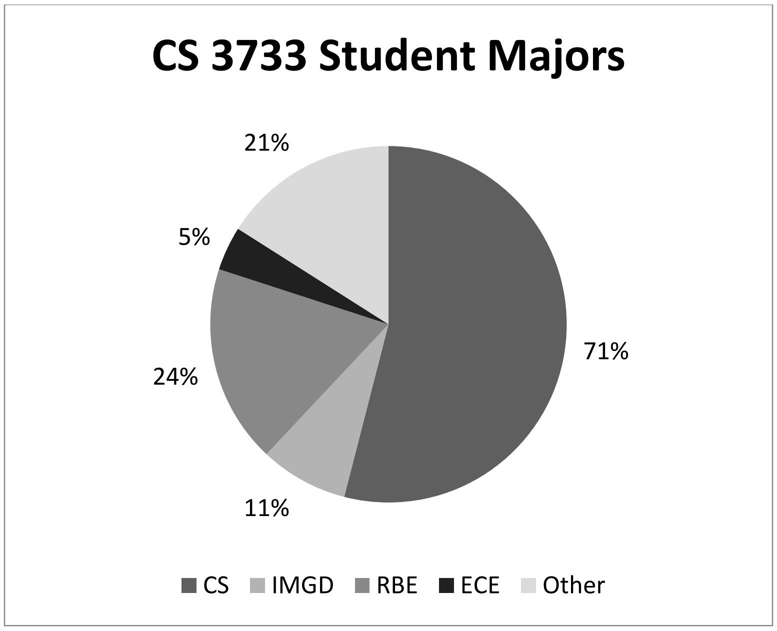


Figure 19: CS 3733 Student Majors

All respondents had previously taken a Java course (specifically CS 2101 Object-Oriented Design Concepts). About 85% had also taken a systems programming course and 25% took an introductory relational databases course. In addition, about 20% of respondents had taken a course on computer networks that covers the basics of the HTTP protocol. A more detailed breakdown of prior knowledge relevant to WPI Suite is shown in Figure 20.

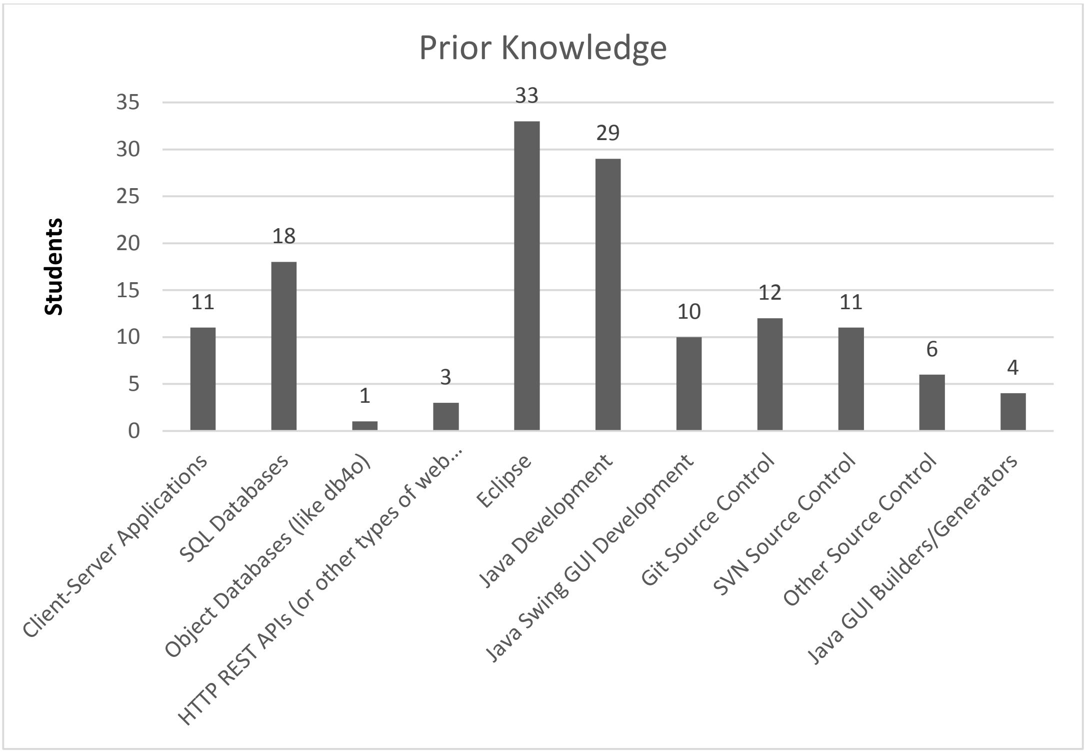


Figure 20: Prior Knowledge of Software Engineering Students

The vast majority of the students indicated they had developed with Eclipse and Java before taking the course. Many also had experience with SQL databases and source control. Very few had used an object database before and only about 10% had previously used an HTTP REST API. Overall, based on the results it is clear that most students were familiar with the fundamental tools necessary to develop for WPI Suite (Java and Eclipse). However, most students did not have a background in object databases, HTTP REST APIs, and Git source control; all of which are central to the WPI Suite project. As will be presented in the next section, students nevertheless were successful in their efforts to develop new modules, which would indicate that the documentation and design of WPI Suite helped them overcome the gaps in their knowledge.

4.1.2 Student Feedback & Confidence

The most important goals of the survey from the perspective of the exemplar module team were to determine the usefulness of the Defect Tracker as an example module, the helpfulness of the accompanying documentation, and the level of student confidence in their understanding of the system. Each of the questions relevant to this goal allowed responses on a scale with three categories. The overall results were very positive, indicating medium to high usefulness of the example module and documentation, and medium to high levels of student confidence.

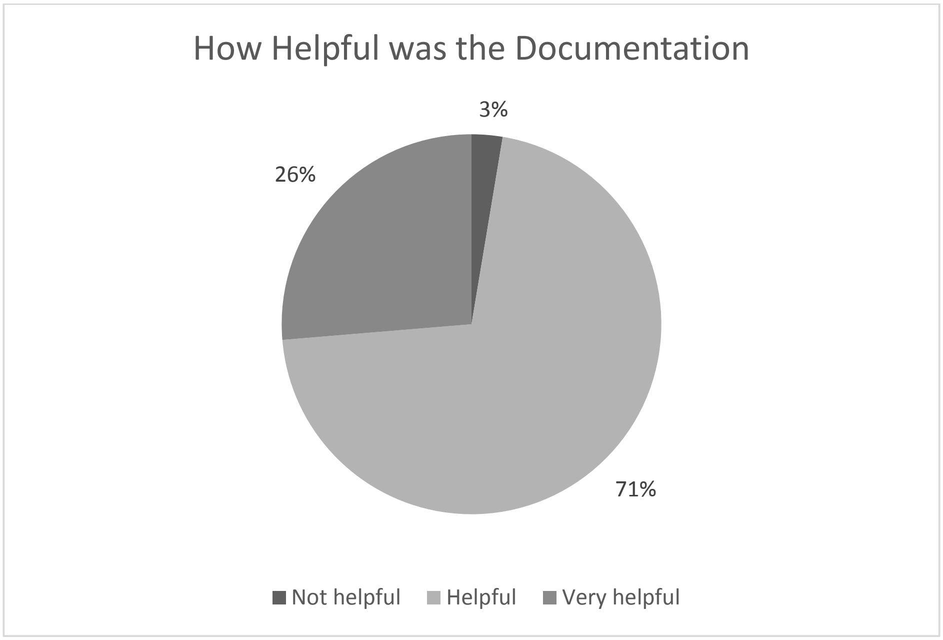


Figure 21: Helpfulness of WPI Suite Documentation

Students were asked to rate the helpfulness of the WPI Suite documentation available on the project's GitHub Wiki. As shown in Figure 21 above, 26% of respondents felt the documentation was very helpful, 71% rated it helpful, and 3% rated it not helpful. An open-ended question elicited feedback such as "[the documentation] is in a well-organized step-by-step structure which is nice" and "incredibly helpful when read by more than one person". Some respondents did indicate the documentation did not do enough to cover frequently encountered problems and that they wished more detail were provided on certain topics. Most of the negative comments were related to setting up the WPI Suite development environment (25% of respondents indicated doing so was difficult).

Feedback on the usefulness of the exemplar module itself was very positive, with 55% of respondents indicating it was very useful and 45% indicating it was somewhat useful. Open-ended feedback indicated students appreciated having an existing module they could use to teach themselves how WPI Suite works. They also felt the code was well documented and helped them get started quickly with their own development. Many were happy that they could reuse existing Defect Tracker code to expedite the development of features for their modules.

While most students were happy with the exemplar module overall, there were some criticisms. The most common were related to getting familiar with components the students had not been exposed to before, including the object database and working with non-blocking Web requests. Complaints about having to implement GUIs in Swing were also mentioned by some. Lastly, some students would have appreciated better documentation of known bugs in both Defect Tracker and the WPI Suite core. In general, most of the negative experiences were related to the constraints and challenges involved with working on a moderately large preexisting system.

The last set of questions focused on determining how confident the students were that they could develop a robust module in their project groups and how confident they were in their ability to develop a module on their own. 71% of respondents were confident their group could implement a robust and extensible module and 24% were very confident. Only 5% were not confident in their group's ability. Figure 22 below shows the breakdown of student confidence in their group.

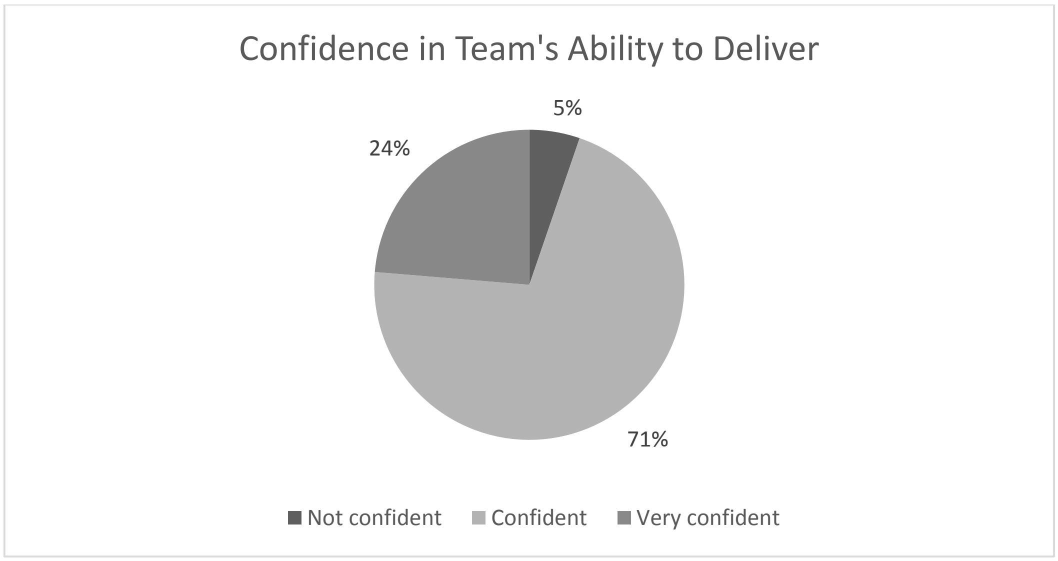


Figure 22: Confidence in Team's Ability to Deliver

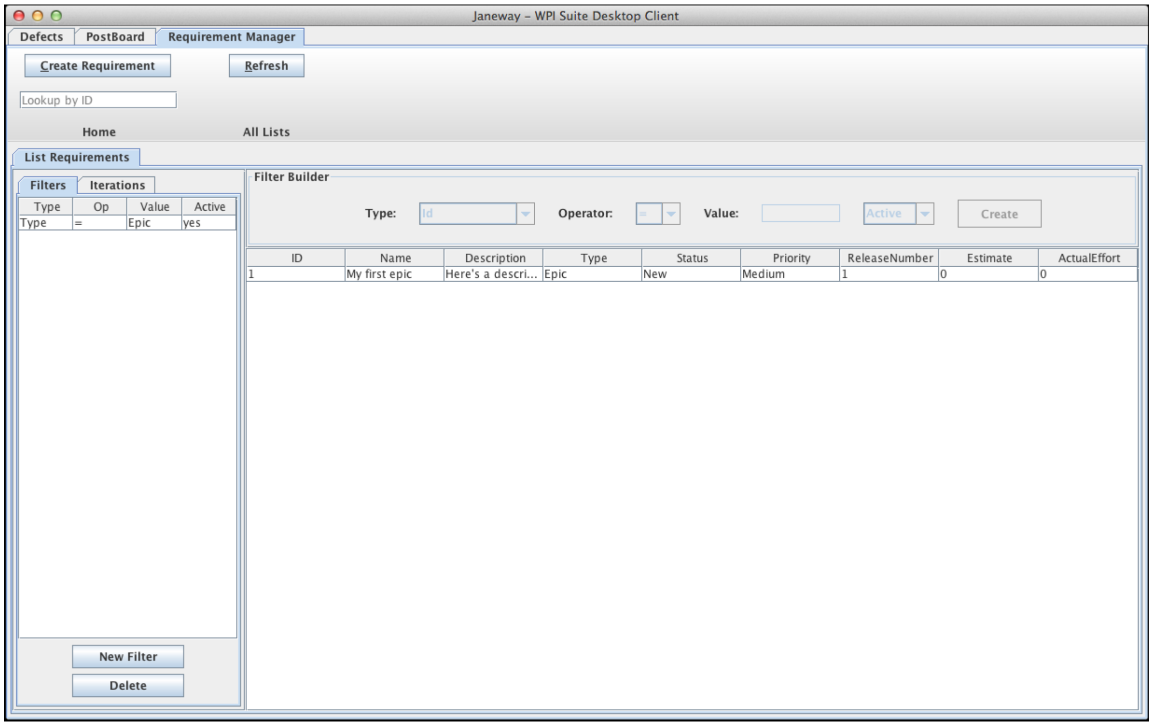
We also asked students how confident they were in their ability to create a new module *on their own*, without help from their team. On the whole, students indicated they were less confident in this regard. Only half were confident and almost 30% were not confident. This variation is likely due to the fact that most of the groups have a couple of more experienced and confident developers that in turn inspire confidence among the group. However, the average student is less confident when faced with the prospect of developing a module individually.

Overall, the majority of the feedback collected indicates that WPI Suite has been well received by students. None of the negative feedback indicated that students were unable to complete the tasks assigned to them in their course. Rather, the criticism focused on the issues that are to be expected from a group of students who have recently had a large, complex software system thrown at them and been asked to understand and improve it.

4.2 Repository Activity

Analysis of the student project groups' GitHub repositories shows how well the teams have been able to pick up and contribute to WPI Suite. The number and quality of features students have managed to add to their modules also provides evidence of the success of WPI Suite. Every D term project team contributed between ten and fifteen thousand lines of code by the end of the fourth week of class. The quality of the code meets our expectations considering students' experience with Java. They demonstrated a basic understanding of design patterns like Model-View-Controller and Observables. Each team also had a high percentage of active members. Most teams had only one person or less (out of about fifteen) who was not making significant code contributions.

All of the project groups made significant progress toward developing a Requirements Manager module. Most of the teams supported creating requirements, assigning requirements to iterations, browsing requirements by iteration, adding notes to requirements, and enforcing user permissions, among an array of smaller features. Figure 23 shows the interface of the requirements manager developed by Team 5.

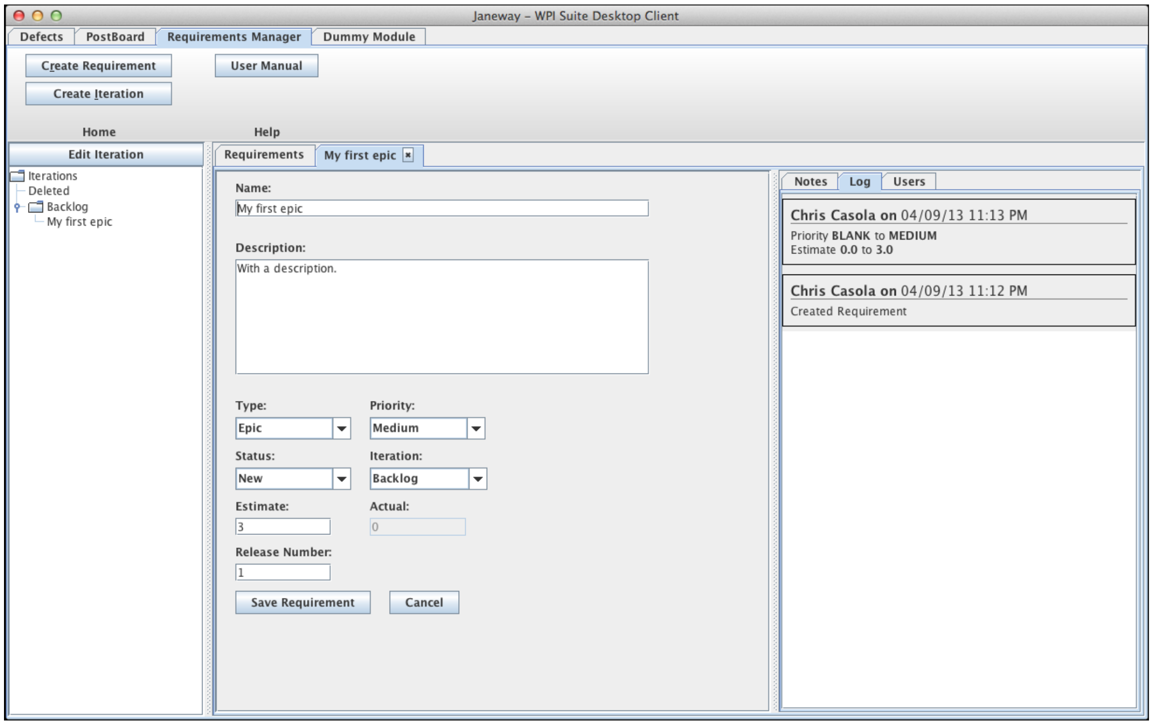


Type	Op	Value	Active
Type	=	Epic	yes

ID	Name	Description	Type	Status	Priority	ReleaseNumber	Estimate	ActualEffort
1	My first epic	Here's a descri...	Epic	New	Medium	1	0	0

Figure 23: Team 5 Requirements Manager Screen Shot

The interface above shows a view that allows users to look through the list of requirements that are currently in the system. On the left there is also a list of Filters that allow users to display only certain requirements based on user-selected criteria. Multiple criteria can be applied and removed to narrow down the requirements that are displayed. Most of the project teams have managed to develop relatively sophisticated and clean user interfaces using the Janeway client. Figure 24 below shows a particularly clean example.



Name: My first epic

Description: With a description.

Type: Epic

Priority: Medium

Status: New

Iteration: Backlog

Estimate: 3

Actual: 0

Release Number: 1

Chris Casola on 04/09/13 11:13 PM
Priority BLANK to MEDIUM
Estimate 0.0 to 3.0

Chris Casola on 04/09/13 11:12 PM
Created Requirement

Figure 24: Team 3 Requirements Manager Screen Shot

There are three main panes in the interface above. The far left displays the various iterations and the backlog. Requirements are shown in a tree structure underneath the iteration they are currently assigned to. There is preliminary support for dragging and dropping requirements among the iterations displayed in the tree. The middle pane shows the requirement fields and the right allows the user to see notes assigned to the requirement and a log of changes to the requirement. Multiple requirements can be open simultaneously in different tabs.

Based on the progress made in the first four weeks of the course, the project teams are clearly proving to be successful in their task to create a requirements module for WPI Suite. The teams already have a significant number of features implemented. Most teams have enough features implemented for their module to be usable as a requirements manager. A focus on improving reliability and stability is underway as of this writing.

4.3 Personal Coaching Experiences

Two of the three exemplar module team members went on to coach teams in the D term software engineering class. They acted as guides and helpful resources for teams who were trying to implement a new WPI Suite module. One subjective way to measure the success of the project is to look at the coaches' personal experiences and see how well the students could work with WPI Suite TNG.

4.3.1 Andrew

Andrew's team managed to get to work extremely quickly. After the team got together and introduced themselves, a single team member was able to add a module to Janeway and get automated builds running on Jenkins (see section 3.1.4 Collaboration Tools) after an evening of work. For the first iteration, they were able to add a simple form to Janeway that could post messages to the server and store entities in the database. This exceeded Professor Pollice's expectations in terms of expected functionality, and getting Jenkins working so quickly was unprecedented. The biggest speed bump was getting the development environment working for all of the team members, especially since there were some unforeseen difficulties on Windows. Admittedly, we should have done more testing on Windows, since most students were on that platform. These issues should be able to be addressed before the next software engineering class begins working.

4.3.2 Chris

Chris's team also got off to a quick start. By the first weekend someone on the team had created a new project for the team's module and implemented the interfaces necessary to build and run a barebones module in the Janeway client. During the first week a lot of time was spent helping all of the team members get WPI Suite running locally on their own machines. Once the team got past that point, active development of a requirements module proceeded quickly. Many of the people on the team proved to be fast learners and did not require handholding at any point. Most of the issues encountered were unrelated to WPI Suite. The team ran into a number of issues caused by their inexperience working on large programming teams. It took some time for team members to adjust to contributing to a large project. They also had to learn how to best use the tools available to them, including the VersionOne requirements manager and GitHub. As of this writing, the team is completely on track and well on their way to delivering a reliable and extensible module that can be improved and maintained by future WPI Suite developers.

4.4 Testing Statistics

The main projects that we worked on were Janeway, Network and Defect Tracker. EclEmma (Mountainminds GmbH & Co. KG and Contributors, 2013), a tool for measuring test case coverage for Java code in Eclipse, was used to determine how much of our code was tested. Our original goal for test case coverage was 80%. Unfortunately, we fell short of this in some areas. The Janeway project has 13% of its code covered by unit tests. The tests that do exist cover all of the `gui.widgets` package and 69.6% of the `gui.container.toolbar` package. The Network project fared better, with 69.6% code coverage. All classes in this package are covered 100% except `Network`, `Request` and `RequestActor`. `Request` and `RequestActor` have 55.4% and 44.1% code coverage, respectively. It is important to improve the code coverage of these two classes as they are the most important parts of the Network project.

The `DefectTracker` project has 26.4% coverage. This coverage is concentrated in the `models`, `entitymanagers`, `validators`, `dashboard` and `search.models` packages. Increasing testing in Defect Tracker is important due to its status as an exemplar module for software engineering students. Students learning from the Defect Tracker module should also be learning of methods for testing the different types of code appearing in the module.

4.5 CodePro Metrics

We used CodePro Analytix, a Java testing tool for Eclipse, in order to measure various metrics regarding our code and also to audit our code (Google, Inc., 2012). For this paper, we measured metrics for commit c57f6adf04[1] on the master branch. Of the metrics that CodePro measures, we decided to include average cyclomatic complexity, average lines of code per method, the average number of parameters and the comments ratio in Table 6. The average cyclomatic complexity measures the number of execution paths in a method. A low number for this metric is beneficial because it makes modifying and debugging code easier. The average lines of code per method metric measures how many lines of code are in each method. It is ideal for this metric's value to be fairly low in order to ensure maximal code reuse and modularity. Having a small number of lines per method also helps developers understand what is going on in a program by breaking down solutions into parts. The average number of parameters metric measures the average number of parameters across all defined methods. A low number for this metric indicates that the methods will be simpler for developers to understand and use and that the methods are not overly complex. The comments ratio metric is calculated by dividing the number of comments by the number of lines of code. The higher the value, the more comments that are included in the code.

[1] This commit can be found on GitHub here: https://github.com/fracture91/wpi-suite-tng/commit/c57f6adf0465bb18ecd4482d07690cbc422ac012

Table 6: CodePro Metrics

	Janeway	**Network**	**DefectTracker**	**Overall**
Avg. Cyclomatic Complexity	1.57	1.60	1.45	1.51
Avg. Lines of Code Per Method	8.07	6.92	7.48	7.50
Avg. Num. of Parameters	0.47	0.42	0.79	0.65
Comments Ratio	21.2%	21.5%	15.3%	17.6%

The number of comments metric allows us to compare the relative quantities of different types of comments. End-of-line comments come after a line of code and sometimes are written to explain the purpose of that code. The multi-line comment is generally used to provide a larger explanation of the purpose of some code. Javadoc comments are used to explain how to use classes, interfaces, methods, and so on and are also useful for generating API documentation for projects. The number of comments metric can be helpful in determining how we have gone about documenting our code and whether or not our code is well documented for new developers. This metric is compared between the Janeway, Network and DefectTracker projects in Figure 25.

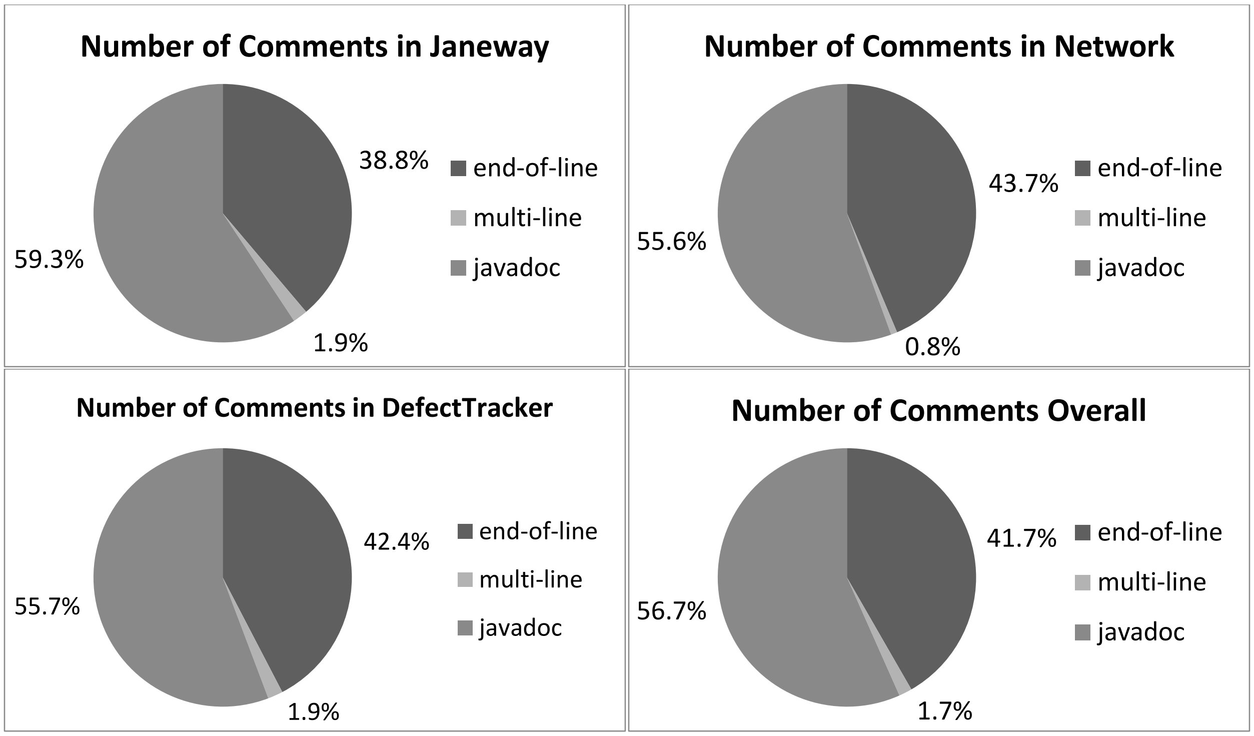


Figure 25: A comparison of the Number of Comments metric

4.6 Problems

4.6.1 JVM Version

One of the most common issues encountered by students when setting up their development environments was being unable to compile and/or run WPI Suite because of an incorrectly installed version of Java. This was exacerbated by the fact that WPI Suite requires a version of Java 7 be installed in addition to version 6 and the majority of students only had version 6. Upgrading to version 7 is not very straightforward, particularly on Windows systems that have existing Java installations. In order to decrease the amount of time spent on dealing with this issue in future classes, the main stakeholders in WPI Suite need to determine which version of Java should be supported. The changes would not be too extensive at this point as far as the WPI Suite core and Defect Tracker, since both were developed to compile to version 6. The Eclipse project settings and ant build files would need to be updated to consistently target one version of Java. This will need to be resolved in future work.

4.6.2 Dynamic Loading and Automatic Eclipse Builds

Another problem is the lack of dynamic module loading in the WPI Suite core. The way entity managers are loaded into the core is through a standard import statement. Module developers need to modify the core source code to instantiate their managers. This introduces a circular dependency between the core project (which needs to import entity managers) and module projects (which need to implement interfaces in the core project). To solve this problem, we moved public interfaces into another project to break the cycle. Still, the build system suffers. While there isn't a cycle as far as Eclipse is concerned, developers need to build the entire project twice: once to build the module JARs into the core project, and again to build the core project itself. This tripped up a number of software engineering students, since the core appears broken unless you build it twice. This issue will need to be fixed by having the core load module JARs dynamically, just like Janeway does.

The build process is awkward for another reason. When JARs change under Tomcat, Tomcat will try to reload them automatically. Since our build scripts build new JAR files into the core project, the reload process is triggered. For some reason, this almost always causes the core server to crash. Even if it did not crash, the core server would most likely subtly break if user sessions in memory were lost, for example. We could not find a way to disable automatic reloading from within the repository, so we had to resort to telling developers to disable automatic builds in Eclipse. This is obviously suboptimal, and should be investigated in the future.

4.6.3 Instructions Are Long

The instructions for setting up the WPI Suite development environment, while thorough, are long. Although these instructions provide a lot of information, following them can be tedious. This may become a barrier to entry for new developers. Aside from simplification of the instructions, providing a virtual machine with a preconfigured development environment may prevent frustration of new developers for WPI Suite.

4.6.4 GUI Testing Framework

Janeway does not currently have tests for the GUI. These tests are important in order to ensure that the GUI remains functional after modifications are made to the underlying code. While it is possible to manually test the GUI to ensure functionality, this method is neither scalable nor efficient. GUI testing allows automation when testing across multiple operating systems. It also saves time on the part of the developer since they don't need to test it. Finally it reduces bugs that may be missed due to a lack of time for testing or a missed step in manual testing. GUI tests may be implemented by firing events in mocked classes and testing the results. Future implementation for these tests is important in order to provide a good example for future WPI Suite developers to follow and learn from.

4.6.5 HTTPS

HTTPS support is essential in order to help prevent passwords from being read from network communications and in order to help prevent session jacking. HTTPS was originally left out to make development easier early on, but was not implemented by the end of the project. To implement HTTPS, modifications will have to be made to the WPI Suite and Network projects. For the Network project, HTTPS capabilities will need to be added to the RequestActor class. The Request class will need to be modified to allow for a developer to pick between using HTTP (in case they want to make requests to non-WPI Suite webservers that do not support HTTPS) and HTTPS.

5 Future Work & Conclusions

After working on the WPI Suite TNG exemplar module, we have identified several subjects for future work. These involve new defect tracking features, overall UI improvements, core API changes, and code reorganization.

Defect Tracker: Ability to search and filter defects

Adding the ability to search for defects and filter the display of defects would significantly increase the usability of the Defect Tracker module. Ideally, users should be able to display only defects assigned to them, only unresolved defects, or any other filter based on the fields of a defect. The filters a user creates should be saved in the database, so that they can be used repeatedly every time the user logs in without needing to recreate them. Users should also be able to search for defects based on text in the title or description fields. Adding this functionality would also make it possible to fully implement a “Dashboard” tab that displays a number of special lists of defects like defects assigned to the current user, new defects in the last 24 hours, or defects reported by the current user that have been recently resolved.

Defect Tracker: Ability to assign defects to milestones

The ability to group defects by assigning them to modules would help project members better manage large amount of defects. A milestone could be created for a planned release or new feature to track the defect associated with that particular item. Deadlines could be set to ensure the bugs associated with the release or iteration are all resolved before the deadline. A similar feature is available on GitHub Issues.

Defect Tracker: Support dependencies between defects

Support for adding dependencies between defects would enable developers to indicate where one defect must be resolved before another. This feature would be useful for larger projects (like WPI Suite) where there are many components and a defect in one component is causing a defect in another.

Janeway: Standardize error-handling mechanisms

The Janeway client does not currently have a built-in API for handling errors and displaying error messages to the user. The Defect Tracker module currently displays error messages using the basic Swing API in each controller where errors could arise. Adding an API for error handling to the Janeway client will help standardize error messages and enable better logging of errors. This API be as simple as a single method with an error level argument, a message argument, and an optional Exceptions that was thrown. Configuration options could then be supplied to the API by the module to specify how, when, and what types of error messages are displayed. Modules could also rely on this API for logging by specifying a low level of error that is not displayed to the user.

Explore alternatives to Swing's SpringLayout layout manager

In both Defect Tracker and Janeway, the `SpringLayout` layout manager was chosen to layout the more complex user interfaces because of the high level of customization and precise control over layout that it provides. While `SpringLayout` is useful in that sense, it is difficult to understand the complicated constraints used to define layouts. Exacerbating the problem is the potential for subtle glitches in the GUI that occur if elements change size or are repositioned, or if the size of the application's window is changed. A possible alternative to `SpringLayout` is the use of simpler layout managers like `FlowLayout` and `BorderLayout` combined with the nesting of `JPanels`. By nesting panels with simpler layouts, complex interface layouts can still be created. The implementation may be slightly longer based on lines of code, but it will likely take significantly less time and be more easily understood by other developers. Other alternatives should also be explored, including identifying ways to improve the existing use of `SpringLayout` and reducing the complexity of existing layout constraints.

Make user interface enhancements to Janeway and Defect Tracker

The user interface needs a few general improvements. Use of icons and color would certainly help users identify buttons and tabs. We designed our interface with icons in mind, and Swing supports adding icons to buttons and tabs, so there is no great engineering burden behind icons. We simply were not able to invest time in icon design. As mentioned in the methodology, we decided to use a standard look and feel across all platforms, but this can be changed in the future to introduce a prettier, native-feeling application. Like any piece of software, the Defect Tracker has many little *paper cut* problems: save buttons should only be enabled after changing something, tabs should indicate changes with an asterisk, etc. Ongoing development will need to focus on these polish issues.

Implement conflict resolution

Currently, if two users edit a defect at the same time, there is no notification of a collision and the last update is the one that ends up in the database. The W3C describes this “lost update problem” and provides a number of solutions (Nielsen & LaLiberte, 1999). We think the most appropriate solution is to condition our updates with the helpful HTTP header “If-Unmodified-Since” (Fielding R. et al., 1999). At the very least, we could warn the user that a change has occurred since they started editing their defect. Ideally, we could provide an interface for displaying and merging in remote changes. This might be accomplished by showing two forms side-by-side, or allowing the user to switch between tabs and determine changes visually. Using the If-Unmodified-Since header would require changes in the EntityManager interface to allow reading the HTTP request headers.

Implement pagination support as a built-in feature of the REST API

In order to work in a production environment tracking thousands of defects, the core and Defect Tracker module will need to support pagination in their APIs. Currently, the desktop client just fetches all existing defects when it needs to list them. If there are two thousand defects in the database, all two thousand will go over the wire and end up in memory in the client. Obviously, this is untenable. Clients need a way to request specific *pages* of defects. It is technically possible to implement pagination through the *advanced* core API, but doing it this way means modules will need to implement pagination on their own, and module HTTP APIs could end up looking very different from one another. Doing pagination in a sane way will require significant work on the database API, the entity manager API, the entity manager implementation itself, and the Janeway UI.

Assess the current build system and recommend improvements

The build process for WPI Suite TNG can probably be improved. Besides the dynamic loading and automatic build problems discussed in the previous section, we cannot help but feel that the recursive Ant solution is suboptimal. Building the entire project can take a full two minutes on slow machines, but most machines take less than thirty seconds. This duration will only get longer as modules are added. We may just be doing something slightly incorrect with Ant, or perhaps the approach is fundamentally flawed. The teams will openly admit to having little experience with Java build systems, so a review by a more seasoned developer may be helpful. One could investigate systems like Maven or Ivy, but one should also make sure that the system is easy enough for students to hook in to.

Improve error handling support in the REST API

The WPI Suite TNG core is currently missing a rather big feature: the ability to pass arbitrary Java objects to the client that represent errors. For example, the Defect Tracker entity manager keeps track of *validation issues*, which contain a field name and an error message associated with that field. However, there is no way to get these issues serialized and sent to the client in a clean way. Consequently, if the user inputs something incorrectly, the client will just see a 4xx error without any further information. It's possible to send back a single *message* string, but this is not sufficient. Work to support this feature is ongoing, and introduces challenges with deserialization since error types may not be known ahead of time.

Refactor to encourage reuse of Defect Tracker code

The project is in need of some refactoring to maintain consistency and encourage code reuse. Some of our packages are structured inconsistently – for example, the defect package is an assortment of classes with no further organization, while the search package is split up into model-view-controller sub packages. We will need to come to a conclusion on how to organize things. Several classes in the Defect Tracker project will need to be pulled out into the core or Janeway projects so module developers can reuse them. `ModelMapper` provides an easy way to copy fields between models. There are a few classes that handle the management of tabs which can be made generic. `ValidationIssue` should be accessible for all modules.

Enable the existing HTTPS functionality in the Core and Janeway

HTTPS support is essential in order to help prevent passwords from being read from network communications and in order to help prevent session jacking. HTTPS was originally left out to make development easier early on, but was not implemented by the end of the project. To implement HTTPS, modifications will have to be made to the WPI Suite and Network projects. For the Network project, HTTPS capabilities will need to be added to the `RequestActor` class. The `Request` class will need to be modified to allow for a developer to pick between using HTTP (in case they want to make requests to non-WPI Suite webservers that do not support HTTPS) and HTTPS.

Automated GUI Testing

The lack of automated GUI testing for the Janeway and Defect Tracker projects reduces efficiency when refining the GUI due to the requirement for developers to manually test that their changes have not adversely affected the application. The lack of automated tests also misses an opportunity to provide a useful unit testing example to students. Automated GUI testing should be added during future development of Janeway and Defect Tracker in order to alleviate these problems. In order to accomplish this, we will have to weigh the pros and cons of directly triggering events in unit tests and testing the results versus using a GUI testing framework. Examples of testing frameworks that may be examined include the Abbot Java GUI Test Framework (Wall, 2011), fest (Welcome, 2013), UISpect4J (Medina & Pratmarty, n.d.), and the Window Licker framework (Project Home, n.d.).

Basic Authentication Header in all requests

Currently, basic authentication is only accepted by the core server for login requests which, upon success, return a session cookie. This cookie must then be sent in any subsequent requests to different parts of the server. The lack of the availability of basic authentication means that clients cannot directly query the core's REST API without first logging in. The addition of basic authentication to the rest of the REST API would allow more types of applications to be developed, such as a simple widget on a phone that might only query one part of the REST API. Addition of this feature would also mean that client developers would not necessarily need to deal with cookie management.

Conclusions

From the experiences of the students in Prof. Pollice's D Term software engineering class, the exemplar module appears to be very helpful. Students were able to begin work on their modules within the first week and over the seven week course were able to implement advanced features in their applications. Of these students, 97% of them found the documentation that was provided by the WPI Suite TNG project to be helpful. One student said in our fourth week survey that, "The breadth of the documentation is great." The student continued, "The main problem is that eclipse has thousands of idiosyncrasies and each computer has a different set up. This made finding a solution for every bug hard, but many of the common ones were addressed." This comment demonstrates a need for a standard development environment to be made available to students. In the future, we should make available a virtual machine or bootstrap script that can be used by students to ensure that they don't have problems with their development environment. Another student stated, with regards to documentation, that there, "can be more about development." It is clear from this comment that we may have not sufficiently documented how to go about using the Core, Janeway and Network library APIs, which we should remedy in the future.

Another asset that would be helpful to the WPI Suite TNG project would be a carefully designed school-agnostic survey. Such a survey could be filled out by students who use WPI Suite at WPI and other schools at or near the completion of their course. This survey could provide valuable feedback with regards to how the project is currently going as well as areas that the project needs to focus more on.

Overall, the desktop client and exemplar module that this project team developed became essential components of WPI Suite TNG. In collaboration with the core team, the project team was able to address the two problems this project aimed to solve: lack of a large software project designed for use in software engineering courses, and lack of software engineering tools meant specifically for students.

6 References

About Jenkins CI. (2013). Retrieved from Jenkins: http://jenkins-ci.org/content/about-jenkins-ci

Atlassian. (2012). *Full Screenshot Tour*. Retrieved October 9, 2012, from Atlassian: http://www.atlassian.com/software/jira/overview/screenshot-tour

Bugzilla. (2012, August 30). *Installation List*. Retrieved 2012, from Bugzilla: http://www.bugzilla.org/installation-list/

Cohn, M. (2004). *User Stories Applied.* Boston: Addison-Wesley Professional.

Crockford, D. (2006, July). *RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON).* Retrieved from IETF Tools: https://tools.ietf.org/html/rfc4627

Driessen, V. (2010, January 5). *A successful Git branching model*. Retrieved from nvie.com: http://nvie.com/posts/a-successful-git-branching-model/

Fielding, R. (1999, June). *Hypertext Transfer Protocol -- HTTP/1.1.* Retrieved from W3C Web site: http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.28

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Retrieved from http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Google Inc. (n.d.). *Project Home*. Retrieved from Google-Gson: http://code.google.com/p/google-gson

Google, Inc. (2012, March 27). *CodePro Analytix User Guide*. Retrieved from Google Developers: https://developers.google.com/java-dev-tools/codepro/doc/

Internet Archive. (2011, June 10). *http://www.edgewall.org/gfx/screenshots/newticket.png*. Retrieved October 9, 2012, from Wayback Machine: http://web.archive.org/web/20110610124455/http://www.edgewall.org/gfx/screenshots/newticket.png

Mantyla, M., & Lassenius, C. (2009). What Types of Defects Are Really Discovered in Code Reviews? *Software Engineering, IEEE Transactions on*, 430-448. Retrieved from http://ieeexplore.ieee.org.ezproxy.wpi.edu/stamp/stamp.jsp?tp=&arnumber=6243919

Medina, R., & Pratmarty, P. (n.d.). *Java/Swing GUI testing made simple!* Retrieved from UISpec4J: http://www.uispec4j.org/

Mountainminds GmbH & Co. KG and Contributors. (2013). *Overview*. Retrieved from EclEmma - Java Code Coverage for Eclipse: http://eclemma.org/

Nielsen, H. F., & LaLiberte, D. (1999, May 10). *Editing the web: detecting the lost update problem using unreserved checkout.* Retrieved from W3C Web site: http://www.w3.org/1999/04/Editing/

Project Home. (n.d.). Retrieved from Window Licker: https://code.google.com/p/windowlicker/

The Apache Software Foundation. (2013). *Welcome*. Retrieved from The Apache Ant Project: http://ant.apache.org/

The Apache Software Foundation. (2013). *Welcome to Apache Maven*. Retrieved from Apache Maven Project: http://maven.apache.org

The Eclipse Foundation. (2013). *About the Eclipse Foundation*. Retrieved from Eclipse: http://www.eclipse.org/org/

Wall, T. (2011). *Getting Started with the Abbot Java GUI Test Framework*. Retrieved from Abbot Java GUI Test Framework: http://abbot.sourceforge.net/doc/overview.shtml

Welcome. (2013, February 13). Retrieved from Fixtures for Easy Software Testing: http://fest.easytesting.org/

7 Appendix A: Survey Data

Class	Count	Percent		
Freshman	0	0%		
Sophomore	26	68%		
Junior	11	29%		
Senior	0	0%		
No Answer	1	3%		
Total	38			
Major	(double majors counted twice)			
CS	27	71%		
IMGD	4	11%		
RBE	9	24%		
ECE	2	5%		
Other	8	21%		
Previous Coursework				
CS 1101/1102	36	95%		
CS 2101	38	100%		
CS 2303	32	84%		
CS 2011	25	66%		
CS 3431	9	24%		
CS 3516	7	18%		
CS 4233	1	3%		
CS 4241	0	0%		
CS 4432	0	0%		
Technologies				
Client-Server Applications	11	29%		
SQL Databases	18	47%		
Object Databases (like db4o)	1	3%		
HTTP REST APIs (or other types of web APIs)	3	8%		
Eclipse	33	87%		
Java Development	29	76%		
Java Swing GUI Development	10	26%		
Git Source Control	12	32%		
SVN Source Control	11	29%		
Other Source Control	6	16%		
Java GUI Builders/Generators	4	11%		
How easy to setup dev. Environment?				

Difficult	9	24%		
Moderate	23	61%		
Easy	6	16%		
How helpful was documentation when getting started as a module developer?				
Not helpful	1	3%		
Helpful	27	71%		
Very helpful	10	26%		
How useful has the defect tracker been to you and your team as an example module?				
Not useful	0	0%		
Somewhat useful	17	45%		
Very useful	21	55%		
How confident are you and your group that the requirements module you are building will be robust and extensible?				
Not confident	2	5%		
Confident	27	71%		
Very confident	9	24%		
Given your current knowledge of WPISuite, how confident are you that you could go off and develop your own module?				
	ALL		CS	
Not confident	11	29%	7	0.259259
Confident	21	55%	16	0.592593
Very confident	6	16%	4	0.148148
How accurately have you been able to estimate work for the requirements manager user stories?				
Not accurately	7	18%		
Accurately	29	76%		
Very accurately	2	5%		

8 Appendix B: GitHub Wiki Documentation

The documentation continues on the next page.

Home

Welcome to the wpi-suite-tng wiki! This wiki contains information for developers as well as anyone looking to use WPI Suite to manage their projects (coming soon).

What is WPI Suite?

WPI Suite is a set of software tools that assist developers with project management tasks. Currently this includes defect tracking. It is a modular application designed to support easy module creation and extension. The client application is built using Java Swing and the server is implemented using Java Servlets. The client and server communicate via a public REST API exposed by the server.

Governance

Policy details regarding the future direction of the project

Governance

Architecture

Architecture Overview An overview of the architecture of WPI Suite, including high-level architecture diagrams and an explanation of the interaction between the various pieces of the system.

Janeway Architecture A more detailed look at the design of the desktop client, Janeway.

Development

For everything related to WPISuite development follow the link to the development page below.

Development

License

Eclipse Public License

Governance

Our mission is to create an extensible, maintainable, well-documented project management system for use as an educational tool that both teaches and facilitates software engineering best practices. This document will outline the governance model for the WPI Suite TNG project that will ensure we can meet our goal.

We will operate as a meritocracy: those with considerable experience will have more sway in the decision-making process, and community members can gain experience by contributing to the project. In the absence of interested community members, Gary Pollice will likely drive future contribution and make necessary decisions.

License

All code is licensed under the Eclipse Public License v1.0, and future contributions will be under this license. Any outside libraries or code included in any contributions must be compatible with the EPL.

Decision-making

Decisions will be made publicly in GitHub issues or pull requests. If you want to change something in WPI Suite TNG, you can open an issue or submit a pull request about the change. If there are no objections to a change after a reasonable amount of time, feel free to go forward with it if you have to power to do so. If there are serious disagreements, the issue should be debated in the comments until a consensus is reached among the interested contributors (or at least those worth listening to based on their merit).

Push access

Some contributors will have permission to push directly to the repository of record (we'll refer to these as "maintainers"). Maintainers obviously have more power than other contributors, and this comes with additional responsibilities and weight in the decision-making process. In order to enable growth, we need a process for outside contributors to become maintainers.

Gaining access

In order to gain push access to the repository of record, a community member will need two existing maintainers to vouch for them. Aspiring maintainers should show a record of good behavior, involvement in the community, trustworthiness, and previous patches that have been accepted into the repository of record. A community member can open an issue and nominate themselves for push access, or an existing maintainer can do so and look for support from one more maintainer. After two maintainers voice their support, the admin will give the new member push access to the repository of record.

Losing access

If a maintainer violates the trust they have been given, their push access can be removed if the rest of the community makes such a decision, as described above. Under emergency circumstances, the admin can immediately revoke push access.

Accepting changes

As described on the Repository Guidelines page, commits that end up on the dev branch will eventually make it into the "released" product on the master branch. As such, these changes should be handled with care.

For contributors without push access, all desired changes must go through a pull request. Any maintainer can accept a pull request and merge it into the dev branch, but they should use their best judgement to determine if they are the best person to handle it considering their familiarity with the affected code, and defer to other maintainers if necessary.

Code review

Changes to the dev branch should undergo a code review before being accepted. This review should make sure that the change:

- does what it says it does
- doesn't introduce regressions
- follows good code style and is programmed sanely
- passes existing unit tests

- contains its own unit tests if necessary

If the pull request doesn't meet these standards, the reviewer may leave comments on the request or on the particular commits stating what's wrong. The requester should then fix problems until the request passes review.

Caveats

Maintainers have the power to change whatever they want, **but they should go through the pull request process as well** and have another maintainer review their changes. Pull requests can be done for branches within the same repository. For small fixes, typos, etc. the maintainer may choose to skip this process - again, they should use their best judgement here and err on the side of caution.

Adding a default module to WPI Suite TNG or large architectural changes should be reviewed by **two** maintainers, and probably have the approval of the community at large.

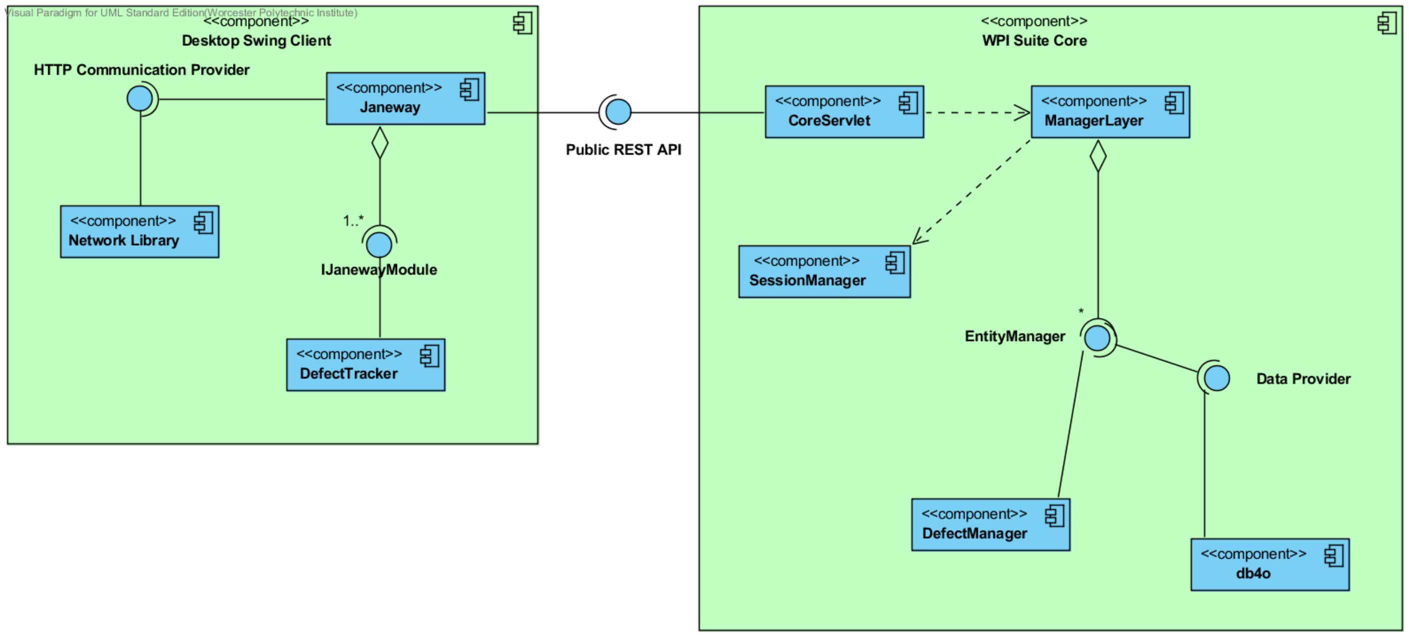
Branches other than dev may choose to adopt some or all of this process. For example, we might start a branch for adding a brand new module that we want to keep clean to avoid reviewing the entire module before merging into dev.

Architecture Overview

The WPI Suite system follows the basic client-server architecture:

- Client
  - used to access projects
  - display module GUIs
- Server (Core)
  - provides user and project management functionality
  - coordinates storage and retrieval of project and module data
  - implements the public REST API

Architecture Diagram



The diagram above shows the main components of the WPI Suite system. These components consist of the following:

- Desktop Swing Client
  - Janeway
  - Network Library
- Core
  - Public REST API
  - Core Servlet
  - Manager Layer
  - Session Manager
  - Entity Managers
  - Database (currently db4o)

Desktop Swing Client

The desktop client is implemented in Java. It consists of a library that is used to generate, send, and manage HTTP requests/responses and sessions as well as a Swing-based application that dynamically loads and displays the client-side GUIs for each WPI Suite module.

Janeway

The swing application that displays module GUIs is known as Janeway. Each WPI Suite module provides a .jar file containing a class that implements the IJanewayModule interface. The Janeway client loads all necessary modules from their .jar files dynamically when the client is run. Each module can make use of the network library to make requests to the server.

Network Library

The network library simplifies the process of creating HTTP requests to send and receive model entities from the core. The library provides a Network factory that can be used to easily create requests. Clients using the library can simply create a request, add an observer to be notified when the response is received, and then send the request. The network library takes care of spawning a new thread to send the request, allowing all HTTP requests to be asynchronous without clients needing to manage threads. More detailed information on the network library can be found on the Networking with janeway page.

Core

The core consists of all the components that manage users and projects, implement the public REST API, handle retrieving and storing models, and persist model data.

Public REST API

The public REST API is used by client applications to communicate with the core. It is accessed using HTTP, so it is accessible by almost any type of client implementation. The primary functions of the API are storing and retrieving models. It also manages login and authentication and project selection. Data is passed back and forth by the API using JSON. This necessitates that models be serialized into JSON before being transferred, and then be deserialized upon being received. This article explains the concept of a REST API very well. For more specific details on the functionality of the WPI Suite API, visit the REST API page.

Core Servlet

put content here

Manager Layer

put content here

Session Manager

put content here

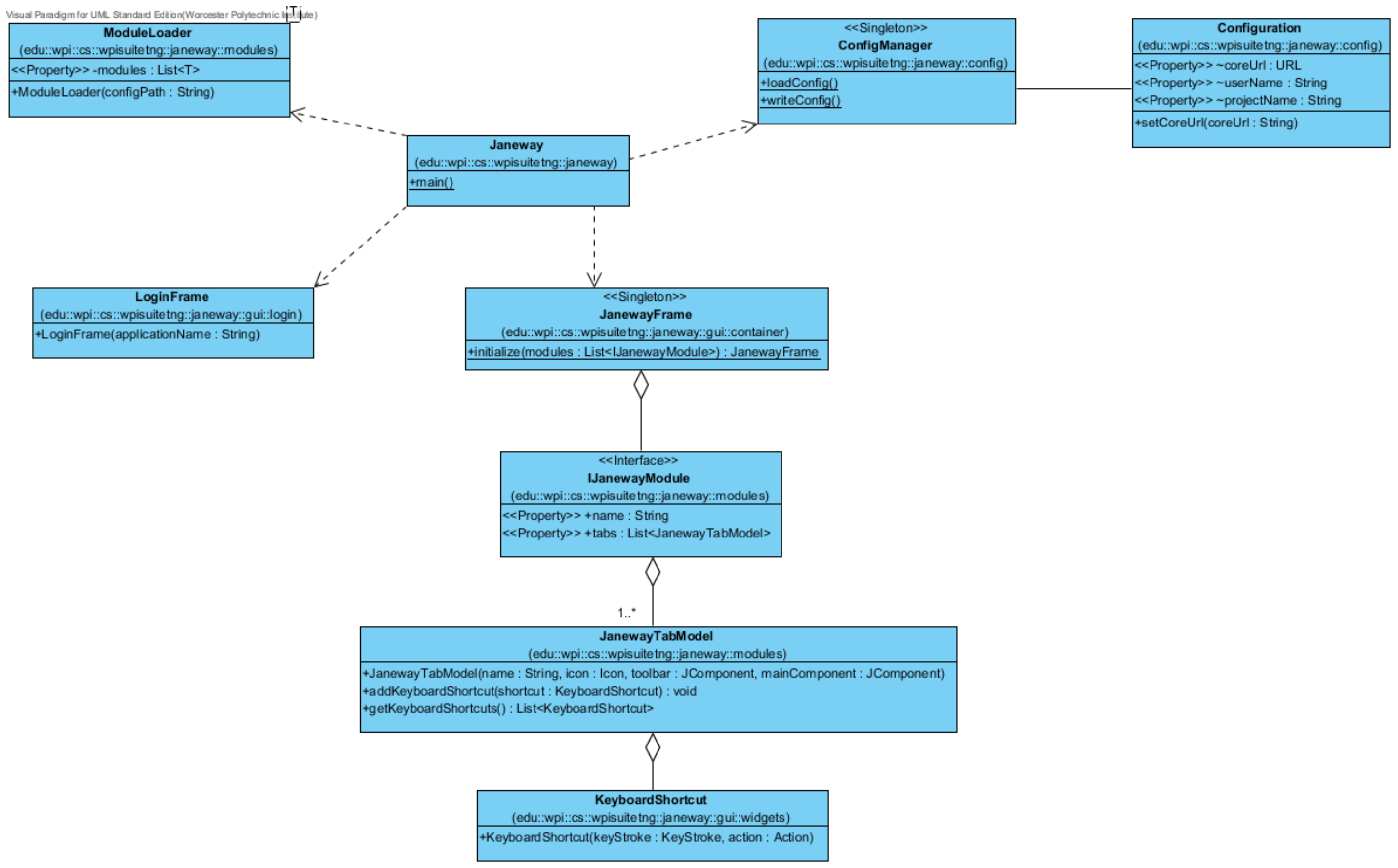
Entity Managers

put content here

Database

put content here

Janeway Architecture


Visual Paradigm for UML Standard Edition(Worcester Polytechnic Institute)
ModuleLoader
(edu::wpi::cs::wpisuitetng::janeway::modules)
<<Property>> -modules : List<T>
+ModuleLoader(configPath : String)
<<Singleton>>
ConfigManager
(edu::wpi::cs::wpisuitetng::janeway::config)
+loadConfig()
+writeConfig()
Configuration
(edu::wpi::cs::wpisuitetng::janeway::config)
<<Property>> ~coreUrl : URL
<<Property>> ~userName : String
<<Property>> ~projectName : String
+setCoreUrl(coreUrl : String)
Janeway
(edu::wpi::cs::wpisuitetng::janeway)
+main()
LoginFrame
(edu::wpi::cs::wpisuitetng::janeway::gui::login)
+LoginFrame(applicationName : String)
<<Singleton>>
JanewayFrame
(edu::wpi::cs::wpisuitetng::janeway::gui::container)
+initialize(modules : List<IJanewayModule>) : JanewayFrame
<<Interface>>
IJanewayModule
(edu::wpi::cs::wpisuitetng::janeway::modules)
<<Property>> +name : String
<<Property>> +tabs : List<JanewayTabModel>
1..*
JanewayTabModel
(edu::wpi::cs::wpisuitetng::janeway::modules)
+JanewayTabModel(name : String, icon : Icon, toolbar : JComponent, mainComponent : JComponent)
+addKeyboardShortcut(shortcut : KeyboardShortcut) : void
+getKeyboardShortcuts() : List<KeyboardShortcut>
KeyboardShortcut
(edu::wpi::cs::wpisuitetng::janeway::gui::widgets)
+KeyboardShortcut(keyStroke : KeyStroke, action : Action)

IJanewayModule

If you are developing a module, all you really need to know about is this interface and JanewayTabModel. If a module wants to display a GUI in the Janeway client, all it must do is provide an implementation of IJanewayModule. The interface requires two methods, getName() and getTabs(). A very simple module can simply return a string from getName() and a list containing one JanewayTabModel for getTabs().

JanewayTabModel

This is a model class that contains data for each tab that is displayed in the Janeway client. Each tab in the client must have a corresponding JanewayTabModel. A module can provide one or more JanewayTabModel objects. A list of JanewayTabModel should be returned by the getTabs() method in your IJanewayModule implementation. JanewayTabModel contains information about a tab, including the name to display on it, an icon for the tab (which can simply be null), the toolbar component, and the main component. The toolbar component is a JComponent (usually a JPanel) that contains the GUI widgets to be displayed in the toolbar of the client. The main component is also a JComponent (usually a JPanel) that contains the main GUI for the module. This is what is displayed in the large portion of the tab. As an example, the defect tracker module returns one JanewayTabModel from the getTabs() method, since it only uses one main tab.

More information about the other classes in Janeway is coming. Module developers should only need to know about the two classes described above. However, there is documentation in the code if you need to know more about the classes not discussed here.

Development

This page contains all of the information you need to get started developing for WPI Suite. Use it to find out how to contribute to the main WPI Suite framework or how to develop your own module to support new functionality.

Setting Up Your Development Environment This guide provides detailed instructions on how to setup your computer to develop for WPISuite.

Building WPI Suite Once you have your development environment setup, follow these instructions to build WPISuite on your machine.

Running WPI Suite TNG Locally These instructions explain how to run WPISuite on your own local Tomcat server. Make sure you have followed the instructions for building WPI Suite before attempting.

Project Layout This page provides a brief description of the purpose of each Eclipse project in the repository.

Repository Guidelines Rules to follow if you want to contribute to the project.

Step by Step Guide to Creating a Module This guide walks you through the process of developing a simple module from beginning to end. It is a good place to start if you want to develop your own module or just want to familiarize yourself with the WPISuite architecture.

Adding a Module This page describes how to add a shell module: the bare minimum classes required of WPISuite modules. If you're just starting out use the step by step guide above.

Networking with Janeway Learn how to make network requests in the Janeway client using the built-in network library.

Adding a Model (Data Entity) This page describes how to add a new data model to a WPISuite module. It primarily focuses on where the model needs to be registered in the core.

Advanced API The specification for the advanced API. This API is a supplement to the standard REST API that allows more advanced requests.

Build System Learn about how the build system works with Eclipse and Ant

WPISuite Troubleshooting & FAQ Find answers to troubleshoot issues and find solutions to common questions.

Setting Up Your Development Environment

The easiest way to develop for WPISuite is to use Eclipse. You can run Eclipse on Windows, Linux, or Mac and easily import all of the projects in the repository. Eclipse also makes it relatively painless to run WPISuite on your local machine.

Table of Contents

Requirements

- Sun JDK 7
- Eclipse IDE for Java Developers with the following plugins:
  - Eclipse Java EE Developer Tools
  - Eclipse Java Web Developer Tools
  - JST Server Adapters
  - JST Server Adapters Extensions
  - EGit
- Tomcat 7
- Git

Install the Sun JDK 7

- On Ubuntu, type `sudo apt-get install openjdk-7-jdk`
- On OS X, follow these installation instructions for installing the JDK 7 from Oracle.
- Windows
  - Go to the Download Page for the Sun JDK 7 and download the installer for Windows x86 (or 64 as long as both your Eclipse installation and Tomcat installation are 64-bit, this is not recommended)
  - Run the installer and accept all of the defaults but **note the path where the jdk is installed**
  - **If you had Java installed previously** do the following:
    - Open the start menu, right-click on Computer, and click Properties
    - Open *Advanced System Settings* from the pane on the left
    - Click the *Environment Variables* button at the bottom of the dialog window
    - Check both the list of user variables and the list of system variables for a variable called JAVA_HOME. If it does not exist click *New* under system variables and create it. For the variable value use the path where you installed the jdk.
    - In the system variables list there should be a variable called PATH. You need to edit this, **do not clear the value**. Simply append %JAVA_HOME%\bin; to the beginning of the value.
    - Your PATH should look something like this now: %JAVA_HOME%\bin; (with other paths after the semicolon)
    - Your JAVA_HOME variable should have something like: C:\Program Files (x86)\Java\jdk1.7.0_17;
    - At this point you should restart your computer. Then open a command prompt window and verify that running the command `java -version` indicates you are running the version of Java that was just installed.

Install Eclipse and Eclipse Plugins

1. Install Eclipse using the link provided above. Choose the version for Java Developers.
2. Once you run Eclipse, it will ask you to choose a workspace location, choose an empty folder on your computer and click OK.

3. Open the Eclipse ***Help Menu*** and click ***Install New Software***.
4. In the ***Work With*** drop-down select the update site for your version of Eclipse.
5. Under the ***Web, XML, Java EE and OSGi Enterprise Development*** section you will find all of the plugins listed above, except Egit. Simply check them off.
6. Next, under the ***Collaboration*** section you will find Eclipse Egit, check that off as well.
7. Click "Finish" to install the plugins.

Note: If Eclipse was **already installed**, but you want to tell it to use the JDK 7, do the following:

1. In Eclipse, select Window > Preferences > Java > Installed JREs
2. Click "Add", choose "Standard VM", and click "Next"
3. Type in the JRE home of the JDK you just installed
   - Linux is usually at /usr/lib/jvm/some-directory-in-here
   - OSX is likely /Library/Java/JavaVirtualMachines/jdk/Contents/Home
   - Windows 7 is at C:\Program Files (x86)\Java\jdk
4. Make sure the system libraries appear in the list box and click "Finish"

Install and Configure Tomcat 7

There are a couple of options for installing Tomcat 7. You can allow Eclipse to download the Tomcat jar and configure itself, or you can manually install Tomcat 7 and then point Eclipse to the install directory.

To take the easy route and let Eclipse get everything setup for you do the following:

1. Open the ***Eclipse preferences window*** (under the Window menu, or the Eclipse menu on Mac systems)
2. Expand the ***Server*** section and then ***Runtime Environments***.
3. Click the ***Add*** button and select ***Apache Tomcat v7.0***.
4. Check ***Create a new local server*** and then click ***Next***.
5. Click the ***Download & Install*** button to automatically download and install Tomcat. Then click ***Finish***.

To manually install and configure Tomcat 7:

- Installing Tomcat on Ubuntu
  1. Install Tomcat 7 (with `sudo apt-get install tomcat7` or from this downloads page)
- Installing Tomcat on OS X
  1. The easiest way is to use the homebrew package manager, if you don't have it you can get it by running this command `ruby -e "$(curl -fsSkL raw.github.com/mxcl/homebrew/go)"`
  2. Do `brew doctor` to verify brew is installed correctly
  3. Run `brew update` to update your local repository
  4. Run `brew install tomcat` to install Tomcat 7 (see these instructions for more help)
  5. The default install directory of Tomcat is /usr/local/Cellar/tomcat/7.xx.xx/libexec
- Install Tomcat on Windows
  1. Download the 32-bit Windows Zip from http://tomcat.apache.org/download-70.cgi
  2. Unzip the file into a convenient location (perhaps C:\Tomcat7)
- Configure Eclipse to use Tomcat on all operating systems Follow the instructions for the "easy route" above. Except instead of clicking "Download and Install", browse for the bin directory inside your Tomcat installation directory.

If you have issues installing and configuring Tomcat, see Fixing Tomcat Issues.

Install Git

In order to work with Git repositories on your local computer, you should install the latest version of Git from the link above. This guide will provide a brief overview on how to use Git, but the Git Book is an excellent resource for future reference.

In order to push from your local repositories to GitHub, you will need to generate and add an SSH key to your GitHub account if you have not already done so. See this page for instructions on how to do this. Alternatively, you can use the Git credential manager and HTTPS repository URLs, which will let you push to GitHub using your GitHub username and password. This might be easier for Windows users, but this guide assumes SSH.

Clone the Repository Onto Your Machine

1. Assuming you have Forked the main repository into your own Github account, you should see a wpi-suite-tng repository in your GitHub account (or you might be using a fork that a teammate set up). Open the repository in GitHub and copy the SSH URL, which should look something like `git@github.com:your_github_account/wpi-suite-tng.git`
2. Open a Terminal - on Windows you'll need to use "Git Bash" which you can find through the start menu
3. Use the `cd` command to change to the directory you want to clone into (e.g. `cd ~/Documents/WPI/SoftEng`).
   - **NEVER** clone your repository into your Eclipse workspace directory, since it can cause problems
4. Type the following command to clone the repository into a new `wpi-suite-tng` directory: `git clone git@github.com:your_github_account/wpi-suite-tng.git`
5. If you type `ls` you should now see a folder called wpi-suite-tng, this is your local git repository
6. Use `cd wpi-suite-tng` to move into the repository folder
7. Type `git checkout master` to check out (for example) the master branch, which you should be on by default

Import Existing WPISuite Projects into your Eclipse Workspace

At this point you'll have all of the source code on your machine, but you'll have to import the Java projects in order to use them with Eclipse. These steps require EGit, which has some nice features for Git/Eclipse integration, but we have run into trouble with using it for day-to-day development and still recommend the command line. You can import the projects without EGit with these instructions.

1. In Eclipse, open ***Window Menu --> Open Perspective --> Other***.
2. Select ***Git Repository Exploring*** and click OK.
3. In the left pane, select the option to ***Add an existing local git repository to this view***.
4. Browse to your workspace directory, click OK, and you should see your git repo listed. Select it and click Finish.
5. Right-click on the ***Working Directory*** folder under your repository hierarchy and click ***Import Projects***.
6. Select ***Import Existing*** and click Next.
7. Select **at least** the following projects to import: Janeway, Network, WPISuite, WPISuite-Interfaces.
   - However, it's probably a good idea to just import all of the projects in the repository
8. Click Finish.

You should now see the projects in your workspace.

Building WPISuite

Building WPI Suite

Run WPISuite locally

Running WPI Suite TNG Locally

Building WPI Suite

This page contains instructions for building WPI Suite on your computer. If you have not already setup your development environment, see Setting Up Your Development Environment.

At this point, there are probably errors in your workspace. They should all be related to missing project dependencies, like .jar files and libraries. To fix them:

1. Open the ***Project*** menu in Eclipse and **uncheck** the option to build automatically.
2. Next, open the ***Project*** menu again and click ***Clean All Projects***.
3. In the dialog that opens, select the options to ***start a build immediately*** and to ***build entire workspace***. Click OK.

After completing the steps above, there may still be errors in your workspace. To fix them:

1. Select the WPI-Suite-Interfaces project in the Eclipse project explorer tab. Right-click on it and click ***Build Project***.
2. Next, right-click on the WPISuite project and click ***Refresh***.

If you have not created a module yet (like the Post Board module), then there will still be some build path errors. To fix this error you need to go into the WPISuite project right click Build Path > Configure Build Path> Libraries and remove the related Jars.

After removing the jar, you will also need to remove it from the entity manager. This can be found under WPISuite>Java Resources> src> edu.wpi.cs.wpisuitetng and Manager layer. You will need to remove the associated import and map.put(MODULE NAME).

All build errors should now be resolved. If they are not, try selecting ***Build All*** from the Project menu. If necessary repeat.

Running WPI Suite TNG Locally

This page contains instructions for running WPI Suite on your computer. If you have not already setup your development environment, see Setting Up Your Development Environment.

You can run both the WPISuite core and the client on your computer. The core runs on the Tomcat server you installed previously.

Before running WPISuite, you need to make sure Eclipse is not set to build before launching, as this will cause issues if you start the core, then run Janeway which triggers a build and overwrites the jars the core has already loaded. To make the change:

1. In Eclipse, open the ***Preferences*** window (Mac: Eclipe --> Preferenes, Other: Edit --> Preferences)
2. Expand the ***Run/Debug*** section and click ***Launching***
3. Under *General Options* **uncheck** the option to "Build (if required) before launching"
4. Click **Apply_** and then ***OK***.

Run the core:

1. Right-click on the **WPISuite** project, select ***Run As***, and click ***Run on Server***.
2. Select the "Tomcat v7.0 Server at localhost" server you created earlier (or define a new server if necessary).
3. Check "always use this server when running this project".
4. Click ***Finish***.
5. After the server starts up, you should see a new tab displaying the "WPI Suite Admin Console". This means the core is running!
6. Login to the core using the default admin account (username: admin; password: password)
7. Create a CoreUser by entering a username, full name, password, and a unique idNum (something greater than 1). Click "Create CoreUser".
8. Create a CoreProject by entering a unique idNum (something greater than 1) and a name. Click "Create CoreProject".

Run the Janeway client:

1. Right-click on the Janeway project, click ***Run As*** and select ***Java Application***.
2. From the popup menu, select the edu.wpi.cs.wpisuitetng.Janeway class and click OK.
3. You should see a login window appear. Enter the username and password for the core user you created.
4. Enter the name of the project you created.
5. The server URL is http://localhost:8080/WPISuite/API if you are running locally with the default configuration.
6. Click Login.
7. You should now see the defects tab in Janeway (or some other module tab).

To stop the core, click the red stop square in the console tab of Eclipse. Whenever you make changes to the projects, you should stop the core, rebuild the workspace, and then restart the core.

Project Layout

WPI Suite TNG is made up of a number of Eclipse projects that are responsible for certain pieces of functionality.

WPISuite

This is the core server project. It provides persistence, an HTTP API for clients, and allows modules to be loaded server-side in order to extend functionality.

Janeway

This is the Swing client that end users can use to interact with the server. The "Janeway" name comes from Captain Janeway, who commands the USS Voyager, just as this client commands the WPI Suite core. Janeway also uses modules to provide extensibility. Janeway modules are loaded dynamically according to the project configuration in the core.

Network

This project is a generalized networking library used in Janeway. It allows modules to make asynchronous HTTP requests in a convenient way. Janeway configures the Network singleton at startup so that modules don't have to worry about the server URL or managing standard HTTP headers. The Network project also provides a convenient dummy server for testing purposes.

Module projects (DefectTracker)

The repository will contain a number of module projects. The only one present at the moment is DefectTracker, but future modules will have a similar format. Modules provide an entry point for both Janeway and the Core. Since they're located in the same project, both components can easily share functionality.

WPISuite-Interfaces

This project currently holds any files used in the core that Janeway or module projects also depend on.

Repository Guidelines

This project's repository will be hosted here on GitHub at https://github.com/fracture91/wpi-suite-tng. The repository is structured using the gitflow methodology. All contributors to this project are expected to follow the rules and best practices described below.

Structure

The repository follows the gitflow methodology as described by Vincent Driessen in his January 2010 blog post, "A successful Git branching model". We will have two main branches in the git repository. They are called dev and master. The dev branch contains the latest changes that will be added to the next release. The master branch contains a tagged revision for each production release. Supporting branches also exist, and for this project they will be called feature and release. A feature branch should be created when adding new functionality and should branch off from the dev branch. Once the feature is fully developed the relevant feature branch should be merged back into the dev branch. A release branch should be created before every production release. The purpose of this branch is to apply finishing touches to the new release while allowing work on the dev branch to proceed with features for the next release.

Master Branch

The release branch should be merged into the master branch only when it is ready to move to production. The revision should then be tagged with the release number.

Rules for committing to the master branch:

1. All commits to the master branch should be the merging back of a release branch that is ready for production.
2. All commits should be tagged with the release number

Dev Branch

The dev branch should branch off the latest production release from the master branch. All development occurs on this branch. When adding a new feature, a new branch should be created off dev. Once work on the feature is complete the feature branch should be merged back into dev. When dev is ready for release, a new release branch should be created that branches off dev. Once the release is ready to move to production, the release branch should be merged back into dev and master branches.

Merges into the dev branch must follow the "Accepting Changes" rules on the Governance page.

Team Branches (temporary)

In order to facilitate cooperation during the MQP, each team will maintain a branch off of the dev branch. These are dev-core and dev-exemplar for the Core team and Exemplar Module team respectively. For each iteration, the teams will work on their own branches to fulfill requirements, pulling in changes from the other team's branch as necessary. Teams can create their own feature branches named dev-core-feature-* or dev-exemplar-feature-*.

At the end of an iteration, changes should be merged into dev as follows:

1. The Exemplar team merges dev-core into dev-exemplar
2. The Exemplar team ensures that tests run and the program is not broken
3. The Exemplar team merges dev-exemplar into dev
4. The Core team merges dev into dev-core at some later point

Feature Branches

Feature branches are used for the development of new functionality or fixing bugs. Feature branches branch off the dev branch. Once the feature has been implemented and is passing all unit tests, it can be merged back into the dev branch (see dev branch merging rules).

Feature branch names should be of the form feature-*

Pushing

Those who have permission to push to the main repository on GitHub must follow the rules below. The master, release, and dev branches must follow these rules strictly. There is some leeway for branches farther away from master, but it is good practice to follow these rules anyway.

Rules for pushing to the main repository:

1. All commits must have an informative commit message in present tense (e.g. "Fix bug in comment editor that prevents adding line breaks")
2. Changes within commits should be focused on a single task. If you're fixing a bug in one file, don't change whitespace in some irrelevant file within the same commit. This is distracting when reviewing commits.
3. Lines should be 110 characters or shorter, except under extreme circumstances

Coding Style

We should look into using code pro for style management.

- If you use a gui-builder for creating a swing gui, you must rewrite its output to be human-readable and understandable
- Indentation
  - Always use tab characters for indentation (no spaces allowed!)
  - Always use spaces for multi-line code alignment
- For non-ternary if statements (ternary if statements are still allowed)
  - always use curly braces
- Curly braces
  - open curly braces should be on the same line as the declaration
  - closing curly braces should be tabbed to match the block declaration start.

Comments

We will be using Javadoc-style Doxygen comments. For each class, a detailed description of the class' functionality must be provided, including class responsibilities. All methods (public, private, and protected) must have a detailed description. They must also have all parameters noted and described with @param (inc. datatype). Finally they must also have all return values and exceptions noted and described.

Naming Conventions

- Class names
  - upper CamelCase
- Method names
  - camelCase (first letter must be lowercase)
  - Prefix getters with “get”
  - Prefix setters with “set”
- Test cases (JUnit)
  - Prefix with “test”

Method Order

1. Fields
2. Constructors
3. Public Methods
4. Protected Methods
5. Private Methods
6. Getters/Setters

Step by Step Guide to Creating a Module

This guide is intended to get you off the ground quickly. It will walk you through creating a very simple module from the beginning. This includes: obtaining a copy of the repository, setting up your development environment, building the entire project, running WPISuite on your local machine, and finally implementing a simple module. The module is called **PostBoard**, and it is basically a message board where anyone can submit a message. The message board is visible to all and anonymous.

If you like, you can download the code for the entire PostBoard module. The most recent code is also available in the WPISuite repository on GitHub. Just checkout the dev-postboard branch.

Table of Contents

If you have not already, you should follow the steps for Forking the Repository, Setting Up Your Development Environment, and Running WPI Suite TNG Locally.

We are going to create a simple module for WPISuite as a demonstration. The module is a post board. Basically, a place where people can write messages that are visible to everyone. The GUI consists of a JList (a listbox), a JTextField, and two JButtons. When the user types a message in the text field and clicks the submit button, it will be displayed in the listbox and sent to the core to be saved. When other users open this module, they will see all of the messages that have been posted. We'll start by creating and configuring a new Eclipse project for this module.

Side note: When starting a new module, it is a good idea to create a new git branch, for this example we'll name it dev-postboard. For more information on repository best practices see the Repository Guidelines.

Create and Configure an Eclipse Project

The project for our module is pretty simple to create. You start by adding a normal Java project. There are a few modifications to make the project build with the rest of the core, but that is about it.

Follow the steps on the Creating a New Module Project page to get started. Make sure to replace all instances of *ModuleName* with the name of the new module we are creating, PostBoard. Skip the last two steps regarding the manifest.txt and modules.conf file, you will be walked through that process later on in this guide.

Implementing a Module

Now that we have a project set up for the PostBoard modulue, we need to start implementing it. The only requirement of WPISuite modules is that they implement the IJanewayModule interface.

1. Using Eclipse's new class wizard, add a class called `PostBoard` that implements `IJanewayModule`. Put it in the `edu.wpi.cs.wpisuitetng.modules.PostBoard` package.

2. Save your new class, you do not need to add any code, and build the entire workspace.

Your module should have built without errors at this point. If there are errors related to dependencies, verify that your build.xml and dependencies.xml files were modified correctly.

Get Your Module to load in the Janeway client

For this step, we'll add the bare minimum to your PostBoard class to allow it to be loaded by Janeway. We'll also add a manifest.txt file to your project so that Janeway knows which class to load for your module.

Modify PostBoard.java:

1. In the `getName()` method, add a return statement to return the name of the module, PostBoard.

```
@Override
public String getName() {
        return "PostBoard";
}
```

2. Next add a field to the class to contain the tabs used by this module. Each module provides a list of tabs that Janeway will display.

```
List<JanewayTabModel> tabs;
```

3. Add a constructor to initialize the list of tabs

```
public PostBoard() {
        tabs = new ArrayList<JanewayTabModel>();
}
```

4. Modify getTabs() to return your list of tabs.

```
@Override
public List<JanewayTabModel> getTabs() {
        return tabs;
}
```

Create a Manifest.txt file:

1. Create a new file in the root of the PostBoard project called manifest.txt
2. Add the following line to the file:

```
module_class edu.wpi.cs.wpisuitetng.modules.PostBoard.PostBoard
```

3. Save the file.

Now you can build the workspace, restart the core, and run Janeway. When you run Janeway you will not see anything to indicate your module exists, but the Janeway console output should indicate that your module was loaded. If it was not, make sure the full name of your module class (including the package) matches the name provided in your manifest.txt file.

Add an Empty Tab to your Module

Now that the PostBoard module is being loaded by Janeway, we would like to add a tab so that the module can display a GUI in Janeway. To do so, we're going to add a simple tab with some placeholder text.

Modify your constructor so that it looks like this:

```
public PostBoard() {
        // Initialize the list of tabs (however, this module has only one tab)
        tabs = new ArrayList<JanewayTabModel>();

        // Create a JPanel to hold the toolbar for the tab
        JPanel toolbarPanel = new JPanel();
        toolbarPanel.add(new JLabel("PostBoard toolbar placeholder")); // add a label with some placeholder text
        toolbarPanel.setBorder(BorderFactory.createLineBorder(Color.blue, 2)); // add a border so you can see the panel

        // Create a JPanel to hold the main contents of the tab
        JPanel mainPanel = new JPanel();
        mainPanel.add(new JLabel("PostBoard placeholder"));
        mainPanel.setBorder(BorderFactory.createLineBorder(Color.green, 2));

        // Create a tab model that contains the toolbar panel and the main content panel
        JanewayTabModel tab1 = new JanewayTabModel(getName(), new ImageIcon(), toolbarPanel, mainPanel);

        // Add the tab to the list of tabs owned by this module
        tabs.add(tab1);
}
```

What the above code does is construct a new JPanel that will contain the toolbar for this module tab (each tab in Janeway has a toolbar section and a main content section). It then adds a JLabel with some placeholder text to the toolbarPanel so that you will be able to identify it. Last, it adds a blue border to the toolbarPanel so you will see where it lies on the screen.

Next, it creates an additional JPanel to hold the main content of the tab. Again, we add a JLabel and a border to identify this panel.

Lastly, we construct a new tab of type JanewayTabModel and pass it the name of the module, an ImageIcon (which could be used to set an icon for this tab), the toolbarPanel, and the mainPanel. Then, we add the new tab to our list of tabs.

If you build your workspace and restart the core and Janeway, you should see a new tab in Janeway called *PostBoard*. On this tab, you should see a blue toolbar area and a green main content area.

Improving the Main Content Panel

Now we want to add some GUI components to our main panel. For the PostBoard module, they consist of a listbox, a textfield, and a submit button.

Add a class called BoardPanel in a view package under your main PostBoard package. The class should extend JPanel. The class will have three fields, a JList, a JTextField, and a JButton.

In the constructor for the BoardPanel class, we need to construct and configure the GUI components. As most of this work is swing-related, the code is not described in detail here, but you can view it below.

```
package edu.wpi.cs.wpisuitetng.modules.postboard.view;

import java.awt.Component;
import java.awt.Dimension;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;

/**
 * This class is a JPanel. It contains the actual post board, a text field
 * for entering a new message, and a submit button for submitting
 * a new message.
 *
 * @author Chris Casola
 *
 */
@SuppressWarnings({"serial", "rawtypes", "unchecked" })
public class BoardPanel extends JPanel {

        /** A list box to display all the message on the board */
        private final JList lstBoard;

        /** A text field where the user can enter a new message */
        private final JTextField txtNewMessage;

        /** A button for submitting new messages */
        private final JButton btnSubmit;

        /**
         * This is a model for the lstBoard component. Basically it
         * contains the data to be displayed in the list box.
         */
        private final DefaultListModel lstBoardModel;

        /**
         * Construct the panel, the three components, and add the
         * three components to the panel.
         */
        public BoardPanel() {

                // Construct the list box model
                lstBoardModel = new DefaultListModel();
                lstBoardModel.add(0, "hello");
                lstBoardModel.add(0, "world");

                // Construct the components to be displayed
                lstBoard = new JList(lstBoardModel);
                txtNewMessage = new JTextField("Enter a message here.");
                btnSubmit = new JButton("Submit");

                // Set the layout manager of this panel that controls the positions of the components
                setLayout(new BoxLayout(this, BoxLayout.PAGE_AXIS)); // components will  be arranged vertically

                // Put the listbox in a scroll pane
                JScrollPane lstScrollPane = new JScrollPane(lstBoard);
                lstScrollPane.setPreferredSize(new Dimension(300,300));

                // Clear the contents of the text field when the user clicks on it
                txtNewMessage.addMouseListener(new MouseAdapter() {
                        public void mouseClicked(MouseEvent e) {
                                txtNewMessage.setText("");
                        }
                });

                // Adjust sizes and alignments
                btnSubmit.setAlignmentX(Component.CENTER_ALIGNMENT);

                // Add the components to the panel
                add(Box.createVerticalStrut(20)); // leave a 20 pixel gap
                add(lstScrollPane);
                add(Box.createVerticalStrut(20));
                add(txtNewMessage);
                add(Box.createVerticalStrut(20));
                add(btnSubmit);
        }
}
```

After saving the BoardPanel class. We need one more class called MainView that also extends JPanel. The purpose of this class is to contain all other JPanel classes that will be displayed in the main content area. The MainView class constructs and sets the arrangement of, the subpanels.

```
package edu.wpi.cs.wpisuitetng.modules.postboard.view;

import javax.swing.JPanel;

/**
 * This panel fills the main content area of the tab for this module. It
 * contains one inner JPanel, the BoardPanel.
 *
 * @author Chris Casola
 *
 */
@SuppressWarnings("serial")
public class MainView extends JPanel {

	/** The panel containing the post board */
	private final BoardPanel boardPanel;

	/**
	 * Construct the panel.
	 */
	public MainView() {
		// Add the board panel to this view
		boardPanel = new BoardPanel();
		add(boardPanel);
	}
}
```

The last thing we need to do to display our new GUI, is to construct the MainView class in our module class's constructor. So we replace the three lines in BoardPanel.java that were constructing the mainPanel with the following:

```
// Constructs and adds the MainPanel
MainView mainPanel = new MainView();
```

Now, if you rebuild your workspace and restart the core and Janeway, you should see your three GUI components on the PostBoard tab.

Add a Button to the Toolbar

To add a button to the toolbar for our PostBoard tab, we need to provide a ToolbarView class that extends JToolBar. This ToolbarView, like MainView, will contain and arrange all subpanels that are part of the toolbar for this module. The MainView class looks like this:

```
package edu.wpi.cs.wpisuitetng.modules.postboard.view;

import javax.swing.JToolBar;

/**
 * This is the toolbar for the PostBoard module
 *
 * @author Chris Casola
 *
 */
@SuppressWarnings("serial")
public class ToolbarView extends JToolBar {

	/** The panel containing toolbar buttons */
	private final ToolbarPanel toolbarPanel;

	/**
	 * Construct this view and all components in it.
	 */
	public ToolbarView() {

		// Prevent this toolbar from being moved
		setFloatable(false);

		// Add the panel containing the toolbar buttons
		toolbarPanel = new ToolbarPanel();
		add(toolbarPanel);
	}
}
```

You'll notice that the ToolbarView class constructs a subpanel of type ToolbarPanel. We need to create this class. Our ToolbarPanel is going to contain one button that refreshes the contents of our post board (the JList in BoardPanel). The ToolbarPanel class looks like this:

```
package edu.wpi.cs.wpisuitetng.modules.postboard.view;

import javax.swing.JButton;
import javax.swing.JPanel;

/**
 * This panel contains the refresh button
 *
 * @author Chris Casola
 *
 */
@SuppressWarnings("serial")
public class ToolbarPanel extends JPanel {

        /** The refresh button */
        private final JButton btnRefresh;


        /**
         * Construct the panel.
         */
        public ToolbarPanel() {

                // Make this panel transparent, we want to see the JToolbar gradient beneath it
                this.setOpaque(false);

                // Construct the refresh button and add it to this panel
                btnRefresh = new JButton("Refresh");
                add(btnRefresh);
        }
}
```

After both ToolbarPanel and ToolbarView have been saved, we need to modify the PostBoard constructor so that it constructs are new ToolbarView. Replace the lines that were constructing a JPanel for the toolbar with the following:

```
// Create a JPanel to hold the toolbar for the tab
ToolbarView toolbarView = new ToolbarView();
```

Then, modify the line that constructs tab1 so that it passes toolbarView to the constructor of JanewayTabModel rather than the old toolbarPanel.

If you build your workspace and restart everything, you should now see a PostBoard tab that has a toolbar and a main area. The toolbar contains a refresh button and the main area contains the list box to hold messages, a text field for entering new messages, and a submit button for submitting messages.

Add Models to Hold the Contents of the PostBoard

The data that must be saved for this module is the messages on the PostBoard. We are going to assume that messages include a String message and a datestamp of type Date. The model for a message is contained in the file PostBoardMessage.java. The class implements the Model interface. This interface is provided by the core, and all models that will be sent to the core must implement this interface.

Two methods that you should take note of are `toJSON()` and `fromJSON()`. The former converts the PostBoardMessage object into a JSON string while the latter converts a JSON string back into a PostBoardMessage. JSON is a method for converting Objects into strings so they can be sent in text form (usually via HTTP). In WPISuite, the Gson library is used to parse objects into JSON strings. These methods are required because they will be used when we start exchanging `PostBoardMessages` with the server.

```
/**
 * Returns a JSON-encoded string representation of this message object
 */
@Override
public String toJSON() {
        return new Gson().toJson(this, PostBoardMessage.class);
}
```

```
/**
 * Returns an instance of PostBoardMessage constructed using the given
 * PostBoardMessage encoded as a JSON string.
 *
 * @param json the json-encoded PostBoardMessage to deserialize
 * @return the PostBoardMessage contained in the given JSON
 */
public static PostBoardMessage fromJSON(String json) {
        final Gson parser = new Gson();
        return parser.fromJson(json, PostBoardMessage.class);
}

/**
 * Returns an array of PostBoardMessage parsed from the given JSON-encoded
 * string.
 *
 * @param json a string containing a JSON-encoded array of PostBoardMessage
 * @return an array of PostBoardMessage deserialzied from the given json string
 */
public static PostBoardMessage[] fromJsonArray(String json) {
        final Gson parser = new Gson();
        return parser.fromJson(json, PostBoardMessage[].class);
}
```

The code above also includes a method called fromJsonArray() which can take as an argument a JSON string containing an array of PostBoardMessage objects and return an actual array of PostBoardMessage objects.

The rest of the methods in PostBoardMessage are required by the Model interface, but are not needed for this module and are therefore left empty or returning null.

In addition to the PostBoardMessage model, we need a model to hold all of the messages in the PostBoard. This model is for client-side use only, so it does not need to implement the Model interface. However, since the Java Swing object we are using to display the PostBoard is a JList, we need to implement an `AbstractListModel` to hold the messages displayed in the GUI.

You can look at the actual PostBoardModel class but the basics are: it contains a field of type `List<PostBoardMessage>` that holds all of the messages, it provides the methods `addMessages(PostBoardMessage[] messages)` and `addMessage(PostBoardMessage newMessage)` for adding messages to the model, and it overrides the `getElementAt(int index)` and `getSize()` methods in the `AbstractListModel` abstract class so that the `JList` can get data from the model.

We need to make one change to the `BoardPanel` class so that it uses our new `PostBoardModel` rather than the `DefaultListModel` we used temporarily. To do this, modify the constructor of `BoardPanel` so that it takes one argument: a `PostBoardModel`. Add a field to store the model and modify the constructor of the `JList` so that the PostBoardModel is provided.

```
public BoardPanel(PostBoardModel boardModel) {

        // Construct the list box model
        lstBoardModel = boardModel;

        // Construct the components to be displayed
        lstBoard = new JList(lstBoardModel);

        ...
```

Make an AddMessageController

Now, we need a controller to handle adding messages to the PostBoardModel when the user clicks the *Submit* button. This controller is very simple. When the button is clicked, it grabs the message from the text field, adds the message to the `PostBoardModel` by calling `addMessage(String newMessage)`, and then clears the text field so the next message can be entered.

```
/**
 *
 */
```

```
package edu.wpi.cs.wpisuitetng.modules.postboard.controller;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import edu.wpi.cs.wpisuitetng.modules.postboard.model.PostBoardModel;
import edu.wpi.cs.wpisuitetng.modules.postboard.view.BoardPanel;

/**
 * This controller responds when the user clicks the Submit button by
 * adding the contents of the message text field to the model as a new
 * message.
 *
 * @author Chris Casola
 *
 */
public class AddMessageController implements ActionListener {

        private final PostBoardModel model;
        private final BoardPanel view;

        /**
         * Construct an AddMessageController for the given model, view pair
         * @param model the model containing the messages
         * @param view the view where the user enters new messages
         */
        public AddMessageController(PostBoardModel model, BoardPanel view) {
                this.model = model;
                this.view = view;
        }

        /*
         * This method is called when the user clicks the Submit button
         *
         * @see java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
         */
        @Override
        public void actionPerformed(ActionEvent event) {
                // Get the text that was entered
                String message = view.getTxtNewMessage().getText();

                // Make sure there is text
                if (message.length() > 0) {
                        // Add the message to the model
                        model.addMessage(message);

                        // Clear the text field
                        view.getTxtNewMessage().setText("");
                }
        }

}
```

In order to begin using this new controller, we need to add it as an `ActionListener` to the submit button in the `BoardPanel` class.

```
// Construct the add message controller and add it to the submit button
btnSubmit.addActionListener(new AddMessageController(lstBoardModel, this));
```

Saving PostBoardMessages to the Core

At this point, you have a version of PostBoard that runs on your computer. Each time you run it, you get a clean PostBoard. You can add messages to it, but they're gone when you restart the application, and others cannot open their copy of the PostBoard application and view them. We're going to fix this by saving messages in the WPISuite core, and then retrieving them to display them in the PostBoard client.

Before proceeding, you should read the Core & Client Communication Overview to gain some background knowledge on communicating with the core.

Creating an *EntityManager*

To allow our PostBoard module to save message in the core we must provide an entity manager for PostBoardMessages. The first step is to create a class that implements the *EntityManager* interface. Each EntityManager is responsible for all requests involving one Model class. The core automatically forwards all requests to the EntityManager. An EntityManager implements the generic `EntityManager<T>` interface.

We will start by creating a new class with the following signature:

```
public class PostBoardEntityManager implements EntityManager<PostBoardMessage>
```

As shown, you must provide the type of the object that this EntityManager will be storing (PostBoardMessage). Eclipse should also automatically generate method signatures for all of the methods required by the EntityManager interface.

You need to provide a simple constructor for the entity manager that takes as an argument a reference to the database so that it can be saved in a field.

```
public PostBoardEntityManager(Data db) {
        this.db = db;
}
```

Next, we need to implement the `makeEntity()` method. This method is responsible for receiving a message in JSON form, parsing it into an actual PostBoardMessage, and then saving it in the database. It must also return the PostBoardMessage that was saved back to the client. It is called whenever a PUT request is received to /postboard/postboardmessage.

```
public PostBoardMessage makeEntity(Session s, String content)
                throws BadRequestException, ConflictException, WPISuiteException {

        // Parse the message from JSON
        final PostBoardMessage newMessage = PostBoardMessage.fromJSON(content);

        // Save the message in the database if possible, otherwise throw an exception
        // We want the message to be associated with the project the user logged in to
        if (!db.save(newMessage, s.getProject())) {
                throw new WPISuiteException();
        }

        // Return the newly created message (this gets passed back to the client)
        return newMessage;
}
```

Lets walk through the code above step-by-step. The first parameter of makeEntity is a Session object containing information about the user making the request. The second parameter is a String containing the body of the HTTP request (in this case the body contains a JSON-encoded PostBoardMessage). The first line of code in the method uses the static method PostBoardMessage.fromJSON to convert the JSON content into a new PostBoardMessage. The if block attemtps to save the PostBoardMessage in the database, throwing an exception if an error occurrs. Importantly, the message is associated with the project that the user is currently logged in to. Finally, the new message is returned. Any exceptions that are thrown within the makeEntity() method are automatically converted into HTTP error codes and sent back to the client. Returning the new PostBoardMessage from the method will cause the core to send the message back to the client.

Now that we are able to save a PostBoardMessage, we would like to be able to retrive them. To do this, we will implement the `getAll(Session s)` method. This method should return all of the PostBoardMessages that have been saved in the database. This method is called when a GET request is received at /postboard/postboardmessage The code for this method is as follows:

```
public PostBoardMessage[] getAll(Session s) throws WPISuiteException {

        // Ask the database to retrieve all objects of the type PostBoardMessage.
        // Passing a dummy PostBoardMessage lets the db know what type of object to retrieve
        // Passing the project makes it only get messages from that project
        List<Model> messages = db.retrieveAll(new PostBoardMessage(null), s.getProject());

        // Return the list of messages as an array
        return messages.toArray(new PostBoardMessage[0]);
}
```

The first line of the method above asks the database to retrieve all objects in the database whose type matches that of the object passed to the retrieveAll method. So using the line `db.retrieveAll(new PostBoardMessage(null), s.getProject())` we are asking the database to return all saved PostBoardMessage objects associated with the current project. The retrieveAll method returns a List containing all of the saved messages. Next, we must return an array of PostBoardMessage, so we need to convert the List to an Array. This can be done using the List class's `toArray()` method.

Next, we need to implement the `getEntity(Session s, String id)` method. This method provides the ability to retrieve a specific PostBoardMessage. Since we will not need to request specific PostBoardMessages in this module, we will make this method throw an exception if a specific message is requested.

```
public PostBoardMessage[] getEntity(Session s, String id)
                throws NotFoundException, WPISuiteException {
        // Throw an exception if an ID was specified, as this module does not support
        // retrieving specific PostBoardMessages.
        throw new WPISuiteException();
}
```

This module will also not be supporting modifying messages, so the `update()` method can simply throw an exception. The same is true for the `deleteEntity()` and `deleteAll()` methods.

```
public PostBoardMessage update(Session s, String content)
                throws WPISuiteException {

        // This module does not allow PostBoardMessages to be modified, so throw an exception
        throw new WPISuiteException();
}
```

There are two methods left to implement, `save()` and `Count()`. The count method is simple, it must simply return the number of PostBoardMessages currently saved in the database.

```
public int Count() throws WPISuiteException {
        // Return the number of PostBoardMessages currently in the database
        return db.retrieveAll(new PostBoardMessage(null)).size();
}
```

The `save()` method is also simple. It must simply save the given PostBoardMessage in the database.

```
public void save(Session s, PostBoardMessage model)
                throws WPISuiteException {

        // Save the given defect in the database
        db.save(model);
}
```

The *EntityManger* for PostBoardMessage is now complete. The full code can be found here.

There is one last step required to let the core know about this new *EntityManager*. We need to modify the `ManagerLayer` class in the core. In the constructor for `ManagerLayer` you should see a few `map.put()` lines. This is where you tell the core to which *EntityManager* requests for a given model should be sent. Add the following line:

```
map.put("postboard" + "postboardmessage", new PostBoardEntityManager(data));
```

The first argument is the concatenation of the name of the module and the name of the model this *EntityManager* is responsible for, the second argument is a new PostBoardEntityManager. The core is now all set to handle requests for the PostBoard model!

Modify AddMessageController to Send Requests

Now that we have EntityManager all set to receive PostBoardMessage objects, we need to start sending it messages whenever they are created by the user in the PostBoard GUI. To do this we will modify the existing AddMessageController so that it also sends a PUT request to the core containing the new message.

In the AddMessageController, remove the line `model.addMessage(message)` which adds messages to the model. Since we will be sending message to the core, we do not want to add them to the model until we are sure they have been saved.

Next, add the following code to send a request to the core that saves the message. This code should be placed beneath the line that clears the text in the new message JTextField.

```
// Send a request to the core to save this message
final Request request = Network.getInstance().makeRequest("postboard/postboardmessage", HttpMethod.PUT); // PUT == create
request.setBody(new PostBoardMessage(message).toJSON()); // put the new message in the body of the request
request.addObserver(new AddMessageRequestObserver(this)); // add an observer to process the response
request.send(); // send the request
```

Line by line, the code above does the following:

1. Use the Network factory to generate a new request object. The request will be sent to the given URL using the given HTTP method.
2. Construct a new PostBoardMessage for the message the user entered, convert the PostBoardMessage object to JSON, and put the JSON string in the body of the request.
3. Add a new observer to the request, this enables us to respond when the response is received (more to come below).
4. Send the request. This call returns immediately without waiting for the response.

The last thing we need to do is add a method to the controller that can be called by the observer when a successful response is received.

```
/**
 * When the new message is received back from the server, add it to the local model.
 * @param message
 */
public void addMessageToModel(PostBoardMessage message) {
        model.addMessage(message);
}
```

At this point there will be an error in the controller, because AddMessageRequestObserver does not exist yet. Proceed to the next section to create it.

Create the AddMessageRequestObserver observer

A RequestObserver is an observer, in other words it "watches" the network request to see when it changes and then takes action based on the change. When a request fails or succeeds, it notifies its observers by calling methods. For a RequestObserver, these methods are `responseSuccess(IRequest)`, `responseError(IRequest)`, and `fail(IRequest, Exception)`. In the AddMessageRequestObserver, we need to implement these methods to take action, particularly when a successful response is received to our request to add a message.

The signature for this class is as follows:

```
public class AddMessageRequestObserver implements RequestObserver
```

The constructor for AddMessageRequestObserver must take one parameter, the AddMessageController. This is necessary so that the `addMessageToModel()` method we created in the previous step can be called when the response is received.

```
public AddMessageRequestObserver(AddMessageController controller) {
        this.controller = controller;
}
```

Next, we need to implement the `responseSuccess()` method. This is where we do most of the work. This method is called automatically when the response is received and it is passed an IRequest. The IRequest provides access to the HTTP response.

```
@Override
public void responseSuccess(IRequest iReq) {
        // Get the response to the given request
        final ResponseModel response = iReq.getResponse();

        // Parse the message out of the response body
        final PostBoardMessage message = PostBoardMessage.fromJson(response.getBody());

        // Pass the message back to the controller
        controller.addMessageToModel(message);
}
```

The comments in the code above are pretty self-explanatory. Basically, we grab the JSON containing the message out of the response body, parse it back into a PostBoardMessage object and then add it to our local model.

The methods that deal with error responses and other failures are below. For this module, we aren't going to do much to notify the user of the errors, but for other modules you would likely want to notify the user somehow.

```
@Override
public void responseError(IRequest iReq) {
        System.err.println("The request to add a message failed.");
}

@Override
public void fail(IRequest iReq, Exception exception) {
        System.err.println("The request to add a message failed.");
}
```

At this point you should be able to save message in the core. You can confirm this by looking at the console output from the core and ensuring your message are being saved when you hit the Save button in the GUI. The next section deals with retrieving message from the core.

Retrieving PostBoardMessages from the core

In order to retrieve messages from the core, we need two more classes, a controller to generate the request and an observer to handle the response. For this module, we will simply be retrieving all messages stored in the core each time a request is sent. The Refresh button on the toolbar will trigger the controller.

Create the GetMessagesController controller

The controller will be named GetMessagesController with the following signature:

```
public class GetMessagesController implements ActionListener
```

The `actionPerformed()` method will be called when the user clicks the Refresh button. This is where we need to send the request to the server for all PostBoardMessages.

```
@Override
public void actionPerformed(ActionEvent e) {
        // Send a request to the core to save this message
        final Request request = Network.getInstance().makeRequest("postboard/postboardmessage", HttpMethod.GET); // GET == read
        request.addObserver(new GetMessagesRequestObserver(this)); // add an observer to process the response
        request.send(); // send the request
}
```

The code above is similar to the request code in the AddMessageController. Basically, we use the Network factory to create a new request, we also use the same path which is the module name plus a / and then the model name. The request type is a GET this time since we want to retrieve models. There is a different observer for this request which we will be creating below.

This controller also needs a method to be called by the observer when it receives messages back from the core in response to our request. This method will take the messages and add them to the local model so that they are displayed to the user in the GUI.

```
/**
 * Add the given messages to the local model (they were received from the core).
 * This method is called by the GetMessagesRequestObserver
 *
 * @param messages an array of messages received from the server
 */
public void receivedMessages(PostBoardMessage[] messages) {
        // Empty the local model to eliminate duplications
        model.emptyModel();

        // Make sure the response was not null
        if (messages != null) {

                // add the messages to the local model
                model.addMessages(messages);
        }
}
```

Create the GetMessagesRequestObserver observer

This observer will be called when responses are received to the requests generated by the GetMessagesContoller. It will be responsible from parsing the JSON array of messages from the response body into an array of PostBoardMessage objects. This observer must also implement the RequestObserver interface. The constructor should also take the GetMessagesController as a parameter so that the `receivedMessages()` method can be called.

```
public class GetMessagesRequestObserver implements RequestObserver
```

The responseSuccess method looks like this:

```
/*
 * Parse the messages out of the response body and pass them to the controller
 *
 * @see edu.wpi.cs.wpisuitetng.network.RequestObserver#responseSuccess(edu.wpi.cs.wpisuitetng.network.models.IRequest)
 */
@Override
public void responseSuccess(IRequest iReq) {
        PostBoardMessage[] messages = PostBoardMessage.fromJsonArray(iReq.getResponse().getBody());
        controller.receivedMessages(messages);
}
```

As you can see above, the static fromJsonArray method in the PostBoardMessage class is used to automatically parse the JSON string in the response body. Once the messages are parsed, the array of messages is passed back to the controller through the `receivedMessages()` method.

To notify users of errors, we are going to provide a slightly more user-friendly implementation of the `responseError()` and `fail()` methods in this observer.

```
/*
 * @see edu.wpi.cs.wpisuitetng.network.RequestObserver#responseError(edu.wpi.cs.wpisuitetng.network.models.IRequest)
 */
@Override
public void responseError(IRequest iReq) {
        fail(iReq, null);
}

/*
 * Put an error message in the PostBoardPanel if the request fails.
 *
 * @see edu.wpi.cs.wpisuitetng.network.RequestObserver#fail(edu.wpi.cs.wpisuitetng.network.models.IRequest, java.lang.Exception)
 */
@Override
public void fail(IRequest iReq, Exception exception) {
        PostBoardMessage[] errorMessage = {new PostBoardMessage("Error retrieving messages.")};
        controller.receivedMessages(errorMessage);
}
```

If an error occurs while retrieving the messages, we will simply put a single message in the JList on the GUI that says "Error retrieving messages".

The last step to enable retrieving messages is to add the GetMessagesController as an actionListener to the refresh button on the ToolbarPanel.

Refactoring

After developing the module, it became apparent that both the views and the controllers would need access to the PostBoardModel. So we will modify the ToolbarView and MainView constructors so that they have one parameter of type PostBoardModel.

The ToolbarPanel class, contained in ToolbarView, also needs access to the model so we add a parameter to the ToolbarPanel constructor as well. This enables us to construct the GetMessagesController inside the ToolbarPanel constructor. The modified constructor looks like this:

```java
public ToolbarPanel(PostBoardModel boardModel) {

        // Make this panel transparent, we want to see the JToolbar gradient beneath it
        this.setOpaque(false);

        // Construct the refresh button and add it to this panel
        btnRefresh = new JButton("Refresh");

        // Add the get messages controller to the button
        btnRefresh.addActionListener(new GetMessagesController(boardModel)); // this is were we needed the model

        // Add the button to this panel
        add(btnRefresh);
}
```

Similar to the ToolbarPanel, our BoardPanel class also needs the model, so we modify the BoardPanel constructor to take one parameter of type PostBoardModel:

```java
public BoardPanel(PostBoardModel boardModel) {
        ...

        // Construct the add message controller and add it to the submit button
        btnSubmit.addActionListener(new AddMessageController(lstBoardModel, this));

        ...
}
```

Some code in the constructor is not included, but the important part where we construct the AddMessageController is shown. The AddMessageController constructor requires the PostBoardModel which is provided, as well as a reference to the view containing the new message field, which is represented by "this".

Finishing Up

At this point your PostBoard module should be fully operational. Feel free to play around and add functionality. Try extending the module to support displaying a user's name next to each message that is posted.

The most recent code for the module can be found in the WPISuite repository, on the dev-postboard branch.

Adding a module

[note: this may be easier in the future if we set up scripts to automate this stuff]

Each module is represented in the repository as a separate Eclipse project. The modules have separate entry points for Janeway and the core. To add a module, follow these steps:

Create an Eclipse project for your module

Follow the steps for Creating a New Module Project.

Provide the required classes for the Janeway client

A module must provide a class that implements the IJanewayModule interface. This interface is located in the Janeway project and it requires two methods, getName() and getTabs(). Basically, it ensures your module class can provide it's name and a list of tabs that it will use.

The tabs are of type JanewayTabModel. A JanewayTabModel provides the name of the tab, an icon, a JComponent (probably a JPanel) that represents the main body of the tab, and a JComponent that holds the toolbar contents for that tab.

For an example module, you can look either at the DummyModule.java file (located in edu.wpi.cs.wpisuitetng.janeway) or for a more complex example, the DefectTracker module. The DummyModule.java is an entire module contained in one file, so it is a simple place to start. You can also follow the Step by Step Guide to Creating a Module.

Creating a New Module Project

This page contains instructions for creating a new Eclipse Project to hold a new WPISuite module. **Note:** In the instructions below, replace *ModuleName* with the name of your new module.

New Java Project

1. Open the File menu in Eclipse and create a new ***Java Project***.
2. Name the project ModuleName.
3. **DO NOT** use the default locations for the project. Your project needs to be inside your local git repository, at the same level as the other projects. The path might be something like ~/SoftEng/wpi-suite-tng/ModuleName.
4. Click ***Next*** and open the ***Projects*** tab.
5. Click ***Add*** and check off the Janeway, Network, and WPISuite-Interfaces projects.
6. Click OK, and Finish.

Configure Build Settings

You need to add a build.xml file to the root of your Eclipse project so that it will build correctly.

1. Copy and paste the build.xml file from the root of the DefectTracker project into your project.
2. ***Right-click*** on the build.xml file in your project and click open with --> text editor.
3. Use Find/Replace in the file to replace all instances of *DefectTracker* with *ModuleName*
4. Save and close build.xml

Next, you need to tell your Eclipse project to use the new build.xml file.

1. ***Right-click*** on your new project in the project explorer and click ***Properties***.
2. Open the ***Builders*** section and click ***New***.
3. Select ***Ant Builder*** as the type and click OK.
4. Under the Buildfile field, click ***Browse Workspace*** and select the build.xml file under your project and click OK.
5. Under the Refresh tab, choose to refresh the entire workspace and subfolders
6. Click Apply to add the new builder.

Finally, you need to modify the dependencies.xml file that is in the root of the git repository.

1. Open dependencies.xml in a text editor and find the `<target name="depend.all"` line. Add to the *depends* list `depend.ModuleName`
2. Next, add a new target at the bottom of the file after the other targets but before the `</project>` tag. The new target should look like this:

```
<target name="depend.ModuleName"
        depends="depend.Janeway">
        <ant dir="${dependencies.basedir}/ModuleName"
                target="${dependency.target}"
                inheritAll="false"/>
</target>
```

3. Finally, modify the depend.WPISuite target so that it depends on your module:

```
<!-- Note that WPISuite depends on DefectTracker because of the lack of dynamic loading -->
<target name="depend.WPISuite"
        depends="depend.WPISuite-Interfaces, depend.DefectTracker, depend.ModuleName"> <!-- depend.ModuleName was added to th
        <ant dir="${dependencies.basedir}/Core/WPISuite"
                target="${dependency.target}"
                inheritAll="false"/>
</target>
```

Modify .gitignore to ignore your project jar file

When your new project is built, it will generate a .jar file that is automatically placed within the core project's lib directory (/Core/WPISuite/WebContent/WEB-INF/lib/ModuleName.jar). You need to tell git not to commit this to the repository.

1. Open the .gitignore file that is in the root of the repository, note this is a hidden file so you might not see it in your directory listing.
2. At the very bottom of the file add a new line with the path to your project's jar file: */Core/WPISuite/WebContent/WEB-INF/lib/ModuleName.jar*
3. Save and close the file.

Add a manifest.txt file to your project

In order for the Janeway client to load your module, you need to provide a manifest.txt file in the root of your project that identifies the class that should be loaded for your module. To do this:

1. Right-click on your project in the Package Explorer, open the New submenu, and click File.
2. Place the file in the root of your module project and name it `manifest.txt`
3. Place one line in the file, which is the directive `module_class` followed by the fully qualified class name of the class in your module that implements IJanewayModule. An example file is below.

```
module_class edu.wpi.cs.wpisuitetng.modules.defecttracker.JanewayModule
```

Modify the modules.conf file in the Janeway project

Note: This step is not necessary if you followed the instructions for configuring your build settings above.

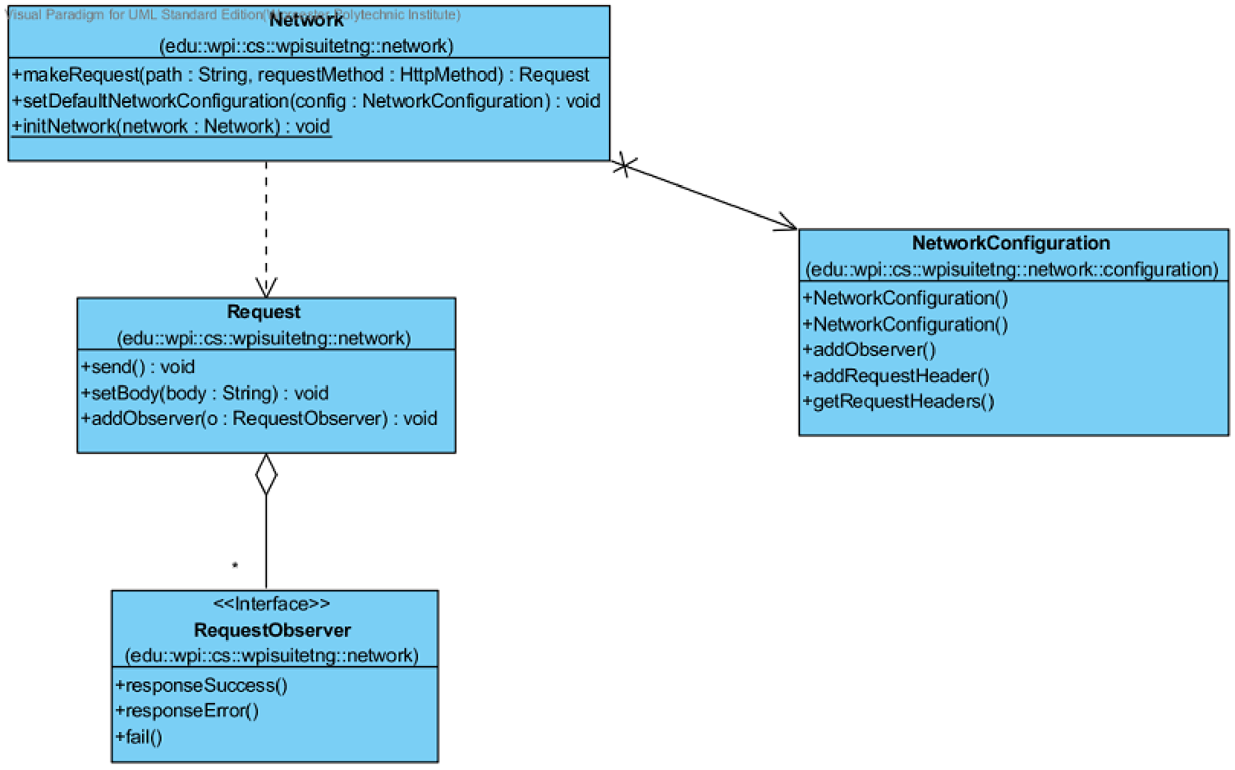
If you modify your project build settings to build the jar file for your project into another location. You need to tell Janeway where to look for the jar. You can add a module directory by doing the following:

1. Open the modules.conf file (located in the root of the Janeway project)
2. If your JAR file is not located in one of the `module_path` directories, add the directory containing it to the list.
3. Save the config file and the next time you launch Janeway your module should load.

Networking with janeway

Janeway communicates with the server by using the Network library, which consists of four key classes:

- Network
- NetworkConfiguration
- Request
- RequestObserver


Visual Paradigm for UML Standard Edition(Worcester Polytechnic Institute)
Network
(edu::wpi::cs::wpisuitetng::network)
+makeRequest(path : String, requestMethod : HttpMethod) : Request
+setDefaultNetworkConfiguration(config : NetworkConfiguration) : void
+initNetwork(network : Network) : void
NetworkConfiguration
(edu::wpi::cs::wpisuitetng::network::configuration)
+NetworkConfiguration()
+NetworkConfiguration()
+addObserver()
+addRequestHeader()
+getRequestHeaders()
Request
(edu::wpi::cs::wpisuitetng::network)
+send() : void
+setBody(body : String) : void
+addObserver(o : RequestObserver) : void
*
<<Interface>>
RequestObserver
(edu::wpi::cs::wpisuitetng::network)
+responseSuccess()
+responseError()
+fail()

The Network Class

The Network class is a singleton class designed to allow Janeway modules to easily create Requests using the default `NetworkConfiguration`, as well as access and change the default `NetworkConfiguration`. `Network`'s `makeRequest` method will construct and return a `Request` which uses the default `NetworkConfiguration`. The `getDefaultNetworkConfiguration` method returns the default `NetworkConfiguration`. This allows you to get the configuration information necessary to construct your own `Request`.

The NetworkConfiguration Class

The NetworkConfiguration class stores information on how Requests should be made. Each instance of NetworkConfiguration stores the URL to the WPISuite API, a set of default request headers, and a set of RequestObservers. Request headers in a NetworkConfiguration, such as cookies, will be sent to the server by any Request using that NetworkConfiguration.

The Request Class

The Request class is used for setting up and initiating a request to the server. By default, all Requests are made asynchronously. This means that Janeway will continue to operate and accept input from the user while the Request is made. A request will be sent to the server after the send method is called.

The RequestObserver Class

The RequestObserver class is used for taking action when a Request completes or fails. They can be added to a Request via the `Request#addObserver(RequestObserver)` method. If a Request is completed successfully, the responseSuccess method of all RequestObservers in that Request will be called. If the Request gets an error from the server, the responseError method will be called. Finally, if the Request is unable to be sent to the server due to a timeout or some other error, the fail method of the RequestObservers attached to the Request will be called.

Adding a Model (Data Entity)

CURRENTLY IN DEVELOPMENT

Model Overview

In WPI Suite TNG, a *Model* is a representation of database contents (or data from another persistent data source). Examples of *Model* data might be: A Defect in the DefectTracker module, a Project in the Core module. *Models* offers persistence and extensible functionality to the data layer.

The classes implementing each *Model*'s attributes live in *edu.wpisuitetng.modules.core.database*.

For more implementation details and other information, visit the Model page [to be written].

How to add a Model

The following is a step-by-step instructional on creating/implementing a new Model. It does not describe the API Adapter implementation.

1. Server (Core Module): Create your Model class

Within the module's /models/ folder, create new class for your Model that implements the *edu.wpi.cs.wpisuitetng.modules.Model* interface. The member variables of the Model class represent the attributes of the data you would like to persist in the database. Each model class should specify an attribute as the Model's 'unique identifier'. For example, the User Model has the 'username' field as its unique identifier attribute.

2. Server (Core Module): Create EntityManager for the Model

Within the the module's /entityManager/ folder, create an implementation of the *edu.wpi.cs.wpisuitetng.modules.core.EntityManager* interface for the Model. The generic fields for the EntityManager correspond to the following: (T) Model Class, (C) Datatype of the Model's unique identifier field.

3. Server (Core Module): Register Model's EntityManager with the ManagerLayer

Data manipulation of the Model class is handled through the EntityManager package. Currently, the EntityManager keeps track of Models using the ManagerLayer map in *edu.wpi.cs.wpisuitetng.modules.core.ManagerLayer*.

3. Client: Create API Adapter for the Model

The API serves the Model classes as JSON strings [format defined in Model.toJSON()]. The client implementation will need to provide an API Adapter for these JSON strings. No standard as of now (10/29/12)

Advanced API

Introduction

The advanced API consists of three module defined methods that were purposefully left to be flexible. The three methods are known as advancedGET, advancedPUT, and advancedPOST.

Syntax

These methods can be accessed by sending the corresponding HTTP request to `/WPISuite/API/Advanced/module/model` where module and model correspond to the module and model you would like to access.

For example, to access the advancedGET functionality of core users, you would send an HTTP GET request to `/WPISuite/API/Advanced/core/user`

Implementation

The function prototypes for the three advanced API methods are:
`public String advancedGet(Session s, String[] args) throws WPISuiteException;`
`public String advancedPut(Session s, String[] args, String content) throws WPISuiteException;`
`public String advancedPost(Session s, String string, String content) throws WPISuiteException;` The implementation of these methods is specific to each module. Suggested usage is listed below.

Advanced GET

The advanced GET function passes the parameters of its URL to the corresponding function. For example, the HTTP GET request `http://example.com/WPISuite/API/Advanced/module/model/arg1/arg2/arg3` would generate a call to the advancedGet method of the the model's manager. The contents of args would be `["module","model","arg1","arg2","arg3",null]`

Advanced PUT

Advanced PUT functions in a very similar way to Advanced GET. The URL string is still broken apart and passed as a String[] to the method implementation.
Advanced PUT also sends the content body of the PUT request as the argument `String content`. **Please note that only the first line of the content body will be read. Anything after the first line break will not be transmitted**

Advanced POST

Advanced POST functions similarly to Advanced PUT, except that where Advanced PUT transmits the entire URL as a String[], Advanced POST only transmits the first argument after `module/model/` For example, in the HTTP POST request `http://example.com/WPISuite/API/Advanced/module/model/arg1/arg2/arg3`. Advanced POST would receive `"arg1"` as it's `String string` argument. **Please note that only the first line of the content body will be read. Anything after the first line break will not be transmitted**

Build System

WPI Suite TNG uses Ant to build projects, following the system described on this page. When using Eclipse to run the software, Eclipse's built-in build system is used, but it is configured to use the Ant scripts when necessary. The build scripts should be able to run without an Eclipse installation, such as in an automated build system.

Usage

Install Ant, then run one of the following commands from a terminal

From the repository root

`ant dist`

Create everything necessary for deployment/release of WPI Suite TNG. This will create a "dist" directory containing a WAR file that can be deployed on a Tomcat server, a runnable JAR for Janeway, and module JARs for Janeway. If you're trying to deploy WPI Suite in a production environment, this is all you need.

`ant compile`

Compile all Java code and copy JARs to appropriate places in order to run WPI Suite in a debug environment.

`ant clean`

Remove all generated resources, including the dist directory and .class files.

`ant test`

Run all JUnit tests. If you're doing this on Jenkins, you should probably use `ant clean test` to make sure your old builds aren't interfering with anything. Another trick (on Linux) is to use `ant test 1>/dev/null` - If nothing is printed out, all of the tests passed since failures are printed to stderr (`1>/dev/null` hides stdout).

From a project directory

Besides any targets defined in the project's buildfile, you should be able to use these targets that all projects inherit

`ant test.single`

Run tests only for this project

`ant compile.deps`

Compile this project and any dependencies

`ant clean.deps`

Clean this project and any dependencies

How it works

If you are unfamiliar with Ant, the basic idea is that you specify a "target" and define the necessary steps to do what's necessary for that target. Targets can depend on other targets, and Ant will fulfill dependencies in the right order for whichever target you're trying to run. For example, you might have a "compile" target which calls the javac compiler in order to produce bytecode that you can actually run.

To summarize the article linked in the intro, each project has its own build.xml file. Project dependencies are defined in a dependencies.xml file. The root build.xml uses dependencies to call project build.xml files in the right order. For example, Janeway depends on Network, so Network must be built beforehand. Projects import build-common.xml, which lets them inherit some functionality and global property definitions. Project buildfiles should override some of the targets specified in build-common.xml.

Adding a module to the build system is covered on the Creating a New Module Project page. For most projects, this will boil down to copying an existing module's buildfile, replacing some names, and telling Eclipse to use it to build JARs into the right places.

Here are the relevant build files and DefectTracker's build.xml as an example:

- https://github.com/fracture91/wpi-suite-tng/blob/master/build.xml
- https://github.com/fracture91/wpi-suite-tng/blob/master/build-common.xml
- https://github.com/fracture91/wpi-suite-tng/blob/master/dependencies.xml
- https://github.com/fracture91/wpi-suite-tng/blob/master/DefectTracker/build.xml

Note that configuration options will need to be duplicated between Eclipse's project settings and the buildfile. For example, project dependencies are specified in both dependencies.xml and your Eclipse project settings. If your Eclipse classpath uses some shared JAR (say, Gson) then you need to make sure to compile with that JAR on your classpath in the buildfile.

WPISuite Troubleshooting & FAQ

For questions and troubleshooting for the Janeway application, see Janeway FAQ.

Q: Where is the server's db4o database file?

A: That is a fantastic question! The db4o database file can be difficult to find -- the location is not consistant between execution methods (Eclipse WTP, Installed Tomcat, JUnit, etc.). The file is consistently named "WPISuite_TNG_local". Below is a list of the locations where the db4o file has been found.

Base of the repository folder : */*
The WPISuite Core Server Project : */Core/WPISuite/*
TOMCAT_HOME (Location of your Tomcat Installation)
JAVA_HOME (Location of your JDK/JRE)
1. *Windows*: Usually: *C:/Program Files/Java/* or *C:/Java/*
2. *OS X*: Usually: */Library/Java/Home/* or */Library/Java/JavaVirtualMachines/jdkx.y.z_aa.jdk/*.
Eclipse Application Folder
1. *Windows* : The location of the *eclipse.exe*
2. *OS X* : Within the *Eclipse.app* file. Right-click the .app > *Open Package Contents*. From here, navigate to *Contents/MacOS/*.
Your home directory on Linux : ~, i.e. */home/username*

Q: When I try to log in, Janeway is returning an error message of "404 - Not Found". When I try to log in via the WebAdmin Console, I receive a "404" error response next to the login button. WHY CAN'T I LOG IN?!

A: This particular response code and message (404, "Not Found") has been encountered when another process has bound itself to your localhost's port 8080. In simpler terms, your Tomcat server is trying to communicate on a port that is already in use by another application.

The solution is to shutdown/close the application that is using the port. There are a few likely culprits: a pre-existing apache (or any other webserver), MatLab application server, etc. Use your process manager to shut down which ever process is using the port.