April 2013

New WPISuite Architecture

Brian Edward Gaffey
Worcester Polytechnic Institute

Michael Perry Della Donna
Worcester Polytechnic Institute

Ryan Christopher Hamer
Worcester Polytechnic Institute

Tyler Michael Wack
Worcester Polytechnic Institute

MQP GFP 1201

New WPI Suite Architecture

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

Michael Della Donna

Brian Gaffey

Ryan Hamer

Tyler Wack

Date: April 25, 2013

Approved:_____

Keywords:
1. Software Engineering
2. WPISuite
3. Client Server Architecture

Professor Gary F. Pollice, Major Advisor

ABSTRACT

WPISuite is a program developed for students enrolled in courses about Software Engineering providing them with an extensible base application that can also be used to manage large development teams and projects. The new design provides a robust, modular, and simplified client-server architecture to access a database. The core was developed using well-documented components and widely used software development patterns to increase ease of use for the software engineering students using it.

ACKNOWLEDGMENTS

We would like to thank Professor Gary F. Pollice, our advisor, for his knowledge, guidance, expertise, and sense of humor throughout this project.

Additionally we would like to thank Christopher Casola, Andrew Hurle, and Jennifer Page for working with our team to leverage our core system to develop the first module for WPISuite: The Next Generation (WPISuite TNG).

Finally we would like to thank the students in Software Engineering D term 2013 for helping test and work with our project as well as designing and building its first student developed module.

Without all of their aid this project would not be possible.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

When working in computer science, a large amount of work is done on teams working on a large code base. Most computer science classes, however, do not mirror this format. The goal of a Software Engineering class is to have students work on a large team to expand a large existing code base. To be able to do this effectively, there must be a program that the students can add to effectively. Additionally, students need access to a wide array of tools such as requirements management, defect tracking, commit logging, and team discussion to name a few. Instructors teaching Software Engineering courses were required to find simple open source projects their students could work on, as well as try and fit industrial project management tools into course work, often with a high learning curve.

This problem led to the development of the original WPISuite. WPISuite provided students with a well-scoped development objective and code base that students could easily work with within a seven-week term. Additionally, WPISuite provided the set of previously mentioned useful tools to students over time. On the whole, the program could serve its purpose for students in WPI's software engineering course initially.

However, the older model of WPISuite became too large, brittle, and unwieldy for students to be able to add to it effectively. Almost forcing the tool into a state of disuse. The goal of this project is to create a new version of WPISuite, a tool that will allow for the simple addition of modules by the students in the Software Engineering class, quickly and easily. Additionally we wanted the new version of WPISuite to be built on common best practices and backed by well-supported open source communities, to ensure it will have a longer life than previous revisions.

The end result of this Major Qualifying Project (MQP) is this report that chronicles the development and development choices of WPISuite: The Next Generation. Additionally we have developed and delivered a working client-agnostic back end system for use in the Software Engineering course beginning in D term 2013, which is also publically available on GitHub to serve as an open source project for the entire Software Engineering educational community.

2 BACKGROUND

To understand the motivation behind WPISuite TNG and our design decisions, this section will elaborate on the history of the previous WPISuite and detail research we performed while designing the system. The architecture and design of WPISuite TNG is motivated by these issues.

2.1 HISTORY

When developing a software project, a large amount of work is done on teams working a large code base. Most computer science classes, however, do not mirror this format. The goal of the Software Engineering class is to have students work on a large team to expand a large existing code base. In Software Engineering, students learn how to work in large groups (~10-15) to build new software functionality into an existing program, and then demonstrate this to the Professor and fellow classmates. Students in the course first began work on Webfoot, an early program that students could add features and functionality to within the seven-week time line of the course.

Students in Professor Pollice's Software Engineering class began work on a productivity suite, WPISuite, in 2007 and have worked on it every year since. WPISuite was first developed to utilize ActiveObjects in order to save data entered into the program and be able to dynamically recreate the objects when loaded. WPISuite functionality increased each term, with a new module being created by multiple teams in each term. The most feature complete and working module would be the final module included in future distributions of WPISuite. However as each module was developed and added into the program, WPISuite became bulky, slow, and hard to understand from a student's perspective. By the end of B term 2011, the program had expanded to over 50,000 lines of code with a multitude of different features and functions including defect tracking, IRC chat, blogging, and commit logging. At this size, how all of the function calls worked and how the program would operate and run were slowly being lost, and harder to grasp and understand enough to able to expand it within one seven week term. In addition, Active Objects did not take off as a standard Java design model and was an elegant yet complicated way of handling objects making it that much harder for students to grasp and understand.

Due to these problems, a new system is being developed, and before implementation of WPISuite began, a considerable amount of thought was placed into two categories, the macro-architecture, the targeted runtime environment, application programming interfaces, and databases.

2.2 MACRO-ARCHITECTURE

After defining some initial project goals and requirements, the team looked into several different macro-architectures to implement the framework.

2.2.1 MVC

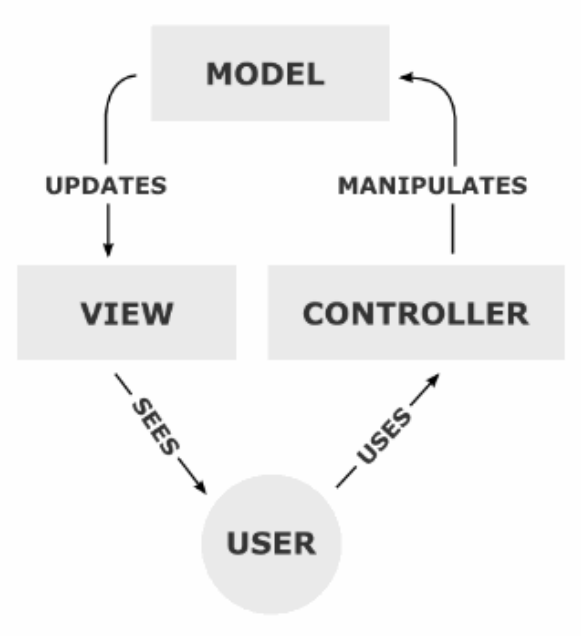

Figure 1 Model-View-Controller Process (Frey, 2010)

The first paradigm that is often suggested for large-scale projects is Model-View-Controller (MVC). MVC is one the most pervasive design patterns in modern software. At a high level, MVC separates decision making into three sections named the model, the view, and the controller. The model represents the data, and is responsible for updating the view. The view is presented to the user, who in turn uses the controller to update the model. This paradigm is very robust and is very useful in large-scale applications. In fact, this pattern was the basis for the previous iteration of WPISuite.

Even though it is very successful in other applications, our team identified several reasons why we should not use this pattern. The existing WPISuite application is a monolithic client side application that only achieves collaboration through a mutual database. Our implementation of the WPISuite is a client server application where the client can be rapidly changed without affecting the server in any way. Model-View-Controller is not the best paradigm for this scenario because the responsibility of updating the view falls to the model. In our proposed implementation this would introduce a complexity in models that we hope to avoid.

2.2.2 PAC

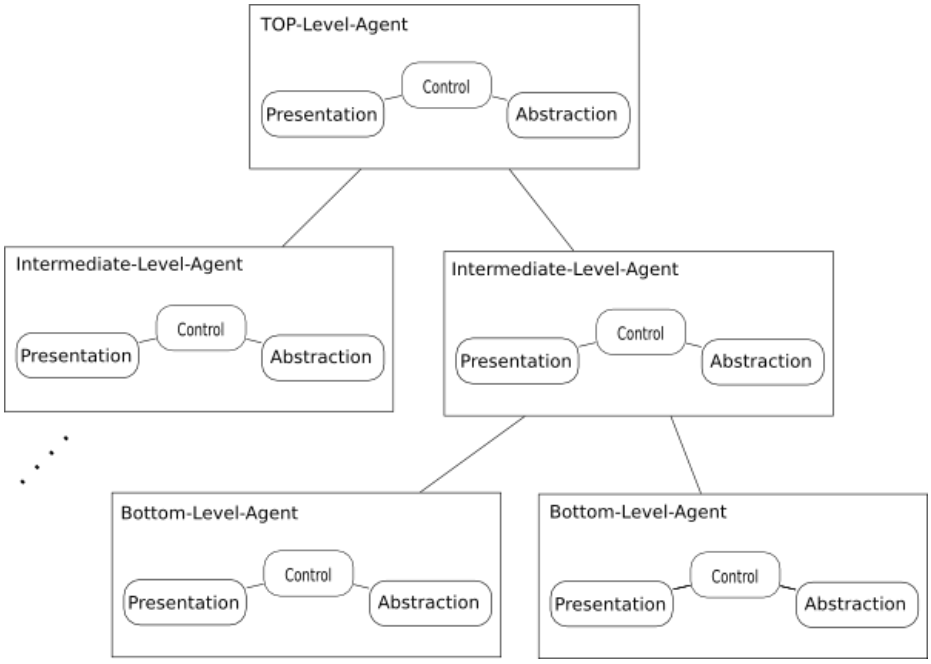

Figure 2 - Presentation-Abstraction-Control Organization ("Presentation-Abstraction-Control Diagram", 2006)

After initially rejecting MVC as an overarching paradigm for WPISuite, the team began a search for a suitable replacement. The next architecture that was discussed by the team was Presentation-Abstraction-Control (PAC). Although PAC is derived from MVC, there are key differences. The Presentation component is responsible for formatting visual presentation of data, the Abstraction component is responsible for retrieving and processing data, and the Control component handles control flow and component communication. This layout makes multithreading easier, because the representation and abstraction are completely insulated from each other. PAC is usually implemented as units called triads each consisting of all three components. This technique is more focused on improving performance with respect to parallel processing and did not match the criteria for a client server based architecture.

2.2.3 MVP

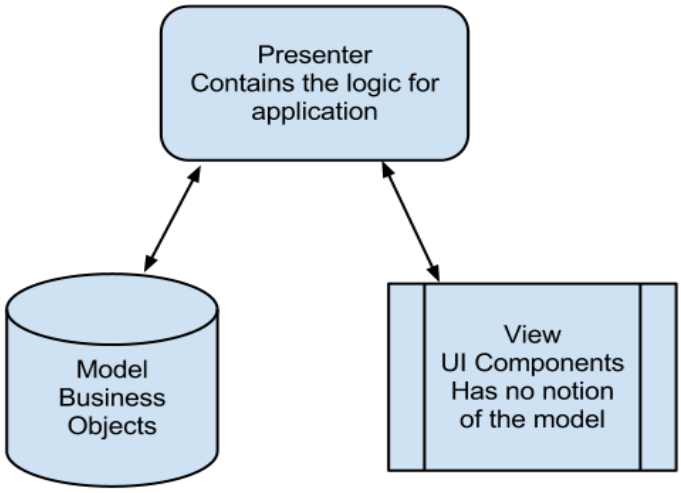

Figure 3 – Model-View-Presenter Model (Cardenas, 2012)

The team eventually settled on Model-View-Presenter (MVP), a close derivative of MVC. Model-View-Presenter replaces the Controller in MVC with a presenter and redefines the original roles of MVC. In MVP, the model only defines the actual data models and does not interact with the view at all. In fact, the model has no concept of a view. The view in MVP serves to display the data, and may have no notion of the models. The presenter interacts with the model to collect data, receives commands from the user forwarded by the view, and sends data back to the view. The team decided on this paradigm because it allowed the model to be completely separated from the view. The architecture that the team designed is not strictly MVP though, because the data sent back from the API assumes that the view has some idea of the data that it requested.

2.3 Runtime Environment Considerations

While the WPISuite core is ultimately written as a Java Servlet that can run inside any Java servlet engine, we targeted one specific runtime in our development and initial deployment. We targeted Apache Tomcat after considering several alternatives and weighing different benefits and costs.

2.3.1 JavaEE vs lightweight Java Server

Our first consideration was whether or not to pick a runtime environment that was fully Java Enterprise Edition (JavaEE) -compliant. After researching the benefits that JavaEE offered, the team felt that they were ultimately unnecessary and well outside both the scale and scope of this project. This did not ultimately rule out JavaEE-compliant web application servers, but the added complexity associated with being JavaEE-compliant was definitely considered a detriment from an education-targeted standpoint. Three JavaEE servers that we considered were Oracle's glassfish, Apache Geronimo, and RedHat jBoss.

These three systems differed very little from each other, but were ultimately rejected based on deployment complexity. (“jBoss Overview”, 2013), (“Java Glassfish”, 2013), (“Apache Geronimo”, 2013)

The next two Web Application Servers that we considered were Apache Tomcat and jetty. Both were lightweight and easy to deploy, going from download to deployed local server. The advantage that led the team to select Tomcat was its ability to render Java Server Pages (JSP). While both servers included an HTTP connector and a servlet engine, jetty lacked the ability to render JSPs. While early development and deployment phases of WPISuite are focused on HTTP communication between a thin graphical client written in Java and the core application framework, future modules and extensions should have the ability to implement web-based access. The ability for Tomcat to both serve as the container for the servlets as well as host and the deliver dynamic web content made it the clear choice for the team.

2.4 Application Programming Interfaces (APIs)

An Application Programming Interface (API) is an interaction layer between software components. APIs expose data, application logic, and other functionality for external software to interact with. For WPISuite, a web API acts as the primary interaction layer between the Database/Server component and the Client component.

Web APIs are web services with control inverted toward the API. Using the API, external software components can leverage the web service. One of the most common web API architectures is REpresentational State Transfer (REST).

2.4.1 REST APIs

REpresentational State Transfer APIs use the HTTP stack to define interactions (Massè, 2012). REST APIs are particularly adept at CRUD layer API interactions because REST maps HTTP verbs to data interactions (see table below). There are many ways to map the HTTP verbs. An example of a REST API’s verb mapping can be seen below.

API Operation	HTTP Request Type
Create	PUT
Retrieve	GET
Update	POST
Delete	DELETE

Figure 4 – WPISuite TNG API HTTP-Operation Mapping

Interactions use Universal Resource Identifiers (URI) to define the data it would like to act on. Since the RESTful API is a web service, the URI is typically in the format of an HTTP URL. For example, a REST API would expose a “student” object with an ID of 42 using the following URL: http://www.wpisuite.com/api/student/42/. To retrieve this object, the HTTP request might look like: GET http://www.wpisuite.com/api/student/42/.

REST APIs have a few inherent advantages:

- **Accessible Interface**: By utilizing the HTTP verb to define interaction, RESTful APIs generate an accessible interface. The accessibility of an interface is especially important to Software Engineering students because it is likely they have little to no experience programming against APIs.
- **Widely Used**: RESTful APIs are a commonly used API architecture. Many resources exist on the Internet to assist students in learning and troubleshooting RESTful API programming.

There are also a number of challenges with REST APIs:

- **Unique Identifier**: The classic REST API uses unique identifier to specify the instance of an entity being interacted with (i.e. the ID integer '42' in /api/student/42/). This creates challenges in database abstraction because many NoSQL databases avoid unique identifiers (primary keys).
- **Format of non-CRUD operations**: REST APIs have defined methods for the CRUD interactions using HTTP verbs. The definition of more complex interactions is less obvious and there are many possible designs.

2.5 Dynamic Module Loading

For WPISuite TNG's Module system to be extensible and modular, the core functionality must support dynamic module loading. In this way, WPISuite TNG can abstract complex data loading and configuration logic that might confuse a Software Engineering student. Ultimately, students should be able to start up a WPISuite TNG client and have the appropriate modules loaded automatically.

There are many possible approaches to designing a dynamic module loading system. The sections below detail a few approaches in use.

2.5.1 OSGi

Open Services Gateway initiative (OSGi) is a specification for software components that supports for dynamic loading. Specified by the OSGi Alliance, OSGi seeks to create modular framework for software by 'bundling' Java components into JAR files with a uniform interface specification. Each bundle is an independent component.

Using OSGi to accomplish dynamic module loading has the following benefits:

- **Widely Used**: OSGi is an established specification with a backing organization (The OSGi Alliance) and many resources. OSGi is implemented by many major software projects, like Eclipse.
- **Eliminates Module Cross-Dependency**: OSGi enforces separation of 'bundles' that eliminates interdependencies between modules. Eliminating interdependency garners benefits in system flexibility because errors in one component will not affect another.

While OSGi is widely used, it is also quite complex from the perspective of a Software Engineering student. The diagram below is a breakdown of different layers in the OSGi model. The complexity of the OSGi system may also make debugging more complicated.

2.5.2 SPRING ROO

Where OSGi is a more low-level approach to dynamic loading, Spring Roo is a high-level, abstracted approach. Developed by SpringSource, Spring Roo is a developer tool that assists with project set up and development. The tool is utilized through a command shell where users issue commands like “perform tests” to run unit tests, or “web mvc setup” to set up a project structure. Roo also supports a number of add-ons that are essentially a wrapper around OSGi.

By abstracting the object loading, Roo offers simplified module loading. By abstracting the OSGi functionality using a wrapper, Roo offers single command module loading. In a Software Engineering class, a Roo-like shell could abstract superfluous module loading details from the student while leaving the logical step.

Despite its benefits, Roo is simply an OSGi wrapper at its core. Developers wishing to code against Roo are still required to adhere to the OSGi specification. This means that complexity is still an issue for Roo.

2.6 DATABASE

WPISuite required a database back end and there were many attributes taken into consideration when choosing database platform to use. The first thing to decide between was whether to use a traditional SQL database or to use an object database. We decided on an object database for two main reasons.

The first reason was the issue of maintainability. We had to consider that this project would be used as a teaching tool for students to develop on. However, predicting what exactly they would be developing would be impractical and time consuming. Since we could not predict what kinds of classes and objects would be created through the development process, we decided an object database would work better. In SQL databases, there are set schemas and to change the schema requires database administrator attention and knowledge, which is not something that we wanted to burden the students with. With an object database, all data is stored in the same database with no separation as SQL databases have with tables. Choosing an object database will allow the students to focus more on their project and give them less overhead during development.

The second reason we chose object databases is the ability of object databases to hold objects in the database instead of having to split up the information into columns of a table. With SQL databases, if the students were creating classes and objects of their own for their module, then they would have to split up the information into types that can be held in those databases. If the students use objects that contain other objects they created as fields, then that can start to get very complicated. With an object database, however, the whole process is more

intuitive. The entire object is stored in the database, even if other objects are fields of the original object. Instead of requiring students to learn what kinds of information can be stored in the database they use and focus on overhead, the object database will allow them to not think about it and just store any object that they need to store in the database.

Once we decided to use an object database, we needed to decide which one to use. When researching object databases, db4o, or DataBase 4 Objects, came to our attention and we found that it had all of the functionality we were looking for. It had everything we needed to be able to abstract the database to the point where students would not need to learn about the internals and how everything worked, the database would just work.

The first aspect of db4o that was thought to be important for simple use by the students was the simple way to change retrieval and deleting depth with a large amount of control. When using an object database, the concern is how far down into an object will things be returned or deleted. Theoretically, one object could hold the entire database and retrieving or deleting that object would retrieve or delete the entire contents of the database respectively. db4o allows for the default depth for both of these things to be changed very easily through the configuration file that is set whenever the database is used. This control can be applied for different classes of objects so that the behavior makes sense for the kind of object that is being interacted with. Some objects it may make sense to delete all of the objects held within it when the original is deleted, such as when the objects are specific to the parent object, and other parent objects hold objects that have multiple functions and are used in multiple places. With db4o's controls for these situations, the team can change the specifications for objects so that they make sense and keep that from being another thing the students have to worry about.

The second thing that db4o had that would allow for easy use by the students was a simple way to connect to it, locally or remotely. Though one of the goals of the project is to abstract the database out as much as possible, there still may be cases when the students have to open it themselves. In those situations it will be very helpful that opening a connection locally or remotely is one simple call. When the students do have to learn some overhead and work directly with the database, db4o makes it simple so the process will be as painless as possible.

3 METHODOLOGY

After researching the previous iteration of WPISuite and different implementation options, we began designing WPISuite TNG. We determined a set of requirements and goals to meet those requirements. Based on these, we made a series of design decisions that affected the final implementation.

3.1 REQUIREMENTS

For WPISuite TNG to be an effective teaching tool for Software Engineering students, there are three main requirements:

1. Provide a framework for development that supports modular extensions.
2. Target framework use cases toward Software Engineering students.
3. Leverage well-documented JVM components as building blocks.

With these requirements in mind, we set out to determine goals for the project and how to meet these goals within the project's time-span.

3.2 SCOPE

The scope of the WPISuite Core was intended to cover creating a deployable application with supporting documentation over a period of 6 months before entering a 2-month period of external beta testing and support. The first month consisted of identifying project requirements and initial architecture design. The next four months consisted of developing the framework. Halfway through that period, the project's feature set was finalized and frozen to facilitate project completion in the required time period. The last month before entering the support phase consisted of final testing and documentation finalization.

The final 2-month period consisted of providing support to the beta testers. The beta testing group was composed of the D term 2013 section of CS 3733 Software Engineering, taught by Professor Pollice.

3.3 GOALS

The goals for the WPISuite Core project are comprised of the following:

- Maintain a level of flexibility allowing the framework to be used across multiple courses.
- Maintain a level of complexity that will allow students with moderate Java exposure to create a complete application in the timeframe of one course.
- Abstract away components that detract from the focus of project management due to the targeted knowledge of the students, specifically in the areas of persistent storage and network communication.
- Maintain a level of documentation that will allow rapid familiarization with the project and setup of a development environment.
- Develop a procedure for accepting submissions as an open source project.

3.4 Problems with the Past

The previous iteration of WPISuite did not have a strict, scalable design from the start. As the application grew, poor design decisions led to code rot as Software Engineering students added code and functionality. The issues in the previous design are summarized here:

- **Lack of effective Client-Server design:** WPISuite was designed as a thick client that interfaced with a SQL database. The extent to which this design was client-server only went so far as the SQL database was on a separate server and the client contacted the SQL server for data. This design relied on client-side processing heavily. It did not take advantage of the server-side to offload processing and improve client-side performance.
- **Poor Login Security**: Security was not one of the original design goals of WPISuite. In later stages, improper login security would reveal a number of vulnerabilities in WPISuite (e.g. passwords left in plaintext).
- **Lack of Module Separation**: As WPISuite grew and modules were added, some modules began to interact. This introduced module dependencies into the WPISuite system without a system to handle dependencies. As such, the modular architecture began to rot and it became difficult to determine which module was responsible for what functionality.

The consequences of these design decisions became apparent as the codebase grew. Poor client-server architecture negatively impacted performance, slowing development. Code complexity complicated development for students. Software Engineering students developing on WPISuite had to spend more time coding/debugging and less time learning project management skills.

3.5 Design

In designing WPISuite TNG, we made a number of design decisions to meet our goals. We also kept the past problems in mind so that we could avoid pitfalls. The following sections go into more detail on the design of the project structure, system modularity, database agnosticism, the API, and security concerns.

3.5.1 Project Structure

The WPISuite Core application is organized into two projects, WPISuite-Core and WPISuite-Interfaces. The WPISuite-Interfaces project consists of various Interfaces and Abstract Classes, along with definitions for the built in models, Project and User. It exists to prevent circular dependencies between WPISuite-Core and modules that might need access to those interfaces.

The WPISuite-Core project is the backbone of the application. It is a Java EE Web project containing three main components. These components are composed of Java HTTP Servlets, an overall controller class and Java Server Pages (JSPs) that implement the admin console.

The Java Servlets handle HTTP communication with the login servlet handling authentication and session management. The admin console is a basic interactive web page implemented as a JSP. It's possible to expand this to hosting other web pages.

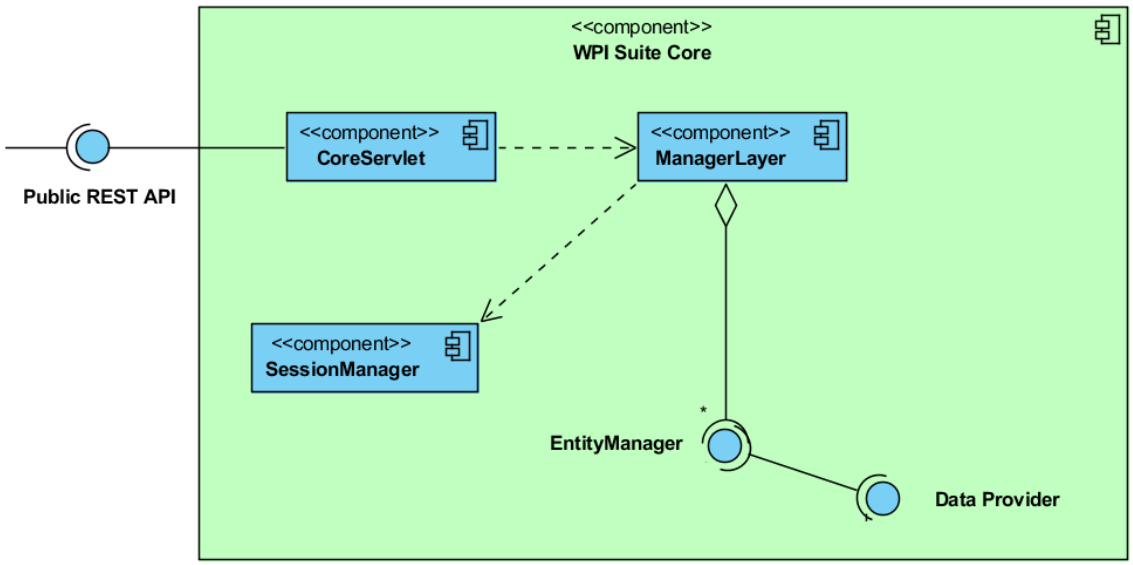

Figure 5 – WPISuite TNG Core Architecture

The main controller or presenter is composed of the ManagerLayer class and supporting EntityManager instances. The ManagerLayer accepts incoming API requests and delegates them to their corresponding EntityManagers. The EntityManagers are registered with the ManagerLayer by each module. A module can have multiple EntityManagers, each representing one Model.

EntityManagers implement a set of seven methods that can be accessed via the API. Four of the commands are standardized to model specific actions like create, read, update, and delete, while the other three are user defined. This system was selected to because it creates an easier mental model for the students due to each method call being associated with it's own type of request.

3.5.2 Modularity

The modularity of WPISuite lies in its module system. WPISuite modules in the core are comprised of at least one EntityManager implementation, usually with the supported Model implementation, though they may contain multiple EntityManagers, Models, as well as supporting classes.

Modules are usually collected into a Java archive (JAR) and bundled into the core web archive (WAR) which can then be deployed to a Java servlet container. Modules are physically associated with the core by instantiating their EntityManager(s) inside the constructor of the

ManagerLayer. After this step, incoming API requests will be automatically routed to the corresponding EntityManager implementation.

This satisfies some of the major goals of the project, such as increasing the flexibility of the frame work by allowing addition and removal of entire modules in a single place, as well as abstracting away network communication by leveraging existing Java HTTP libraries and automatically routing incoming API calls to their appropriate handler methods.

3.5.3 *Database Agnosticism*

WPISuite TNG's data persistence layer was designed to be implementation agnostic by enforcing a Data interface. The interface abstracts the complexity of database interaction into CRUD (Create, Read, Update, Delete) operations, along with a few useful extensions. The simplified database interaction was made to be easier for Software Engineering students to understand and develop with.

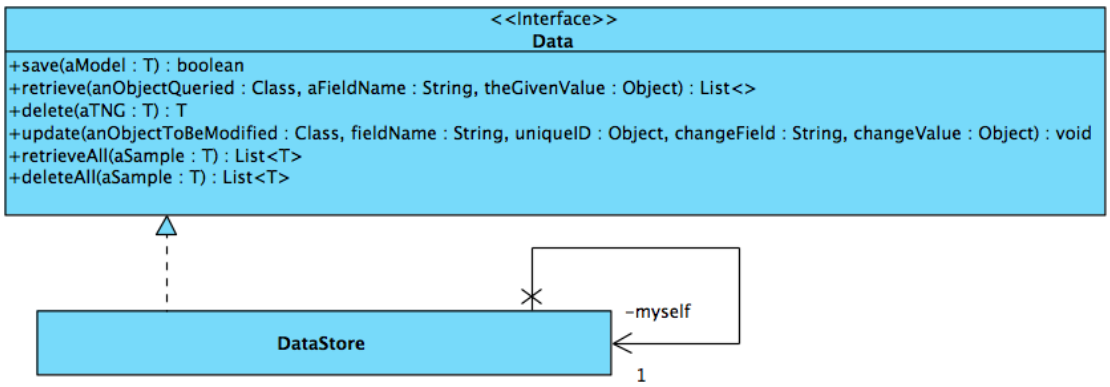

Figure 6 – WPISuite TNG Data Interface

The Data interface also introduces a point of flexibility in the system. Future developers could easily code implementations for MySQL, MongoDB, or other database technologies.

db4o Implementation of Data Interface

We chose db4o as the original implementation of the Data interface for WPISuite TNG (*DataStore*). This implementation is relatively straightforward because db4o inherently supports database entity to object mapping. In this way, we avoided the need for a separate Object Relational Mapping layer on top of the Data interface implementation. db4o also proved to be a flexible implementation of the Data interface because, like the interface, db4o supports querying against any field. Likewise, the Data interface's retrieval method supports querying across any field by default.

Additionally, having a system that holds entire objects made it simpler for the students to understand how to store what they had to store. Instead of having to deal with the complexities

of storing objects that have objects inside of them and BLOB columns with a SQL database, the students did not have to worry about those concerns due to the choice of the db4o database.

3.5.4 API AND COMMUNICATION

The WPISuite core was designed to be client agnostic. The decision was made to support multiple educational opportunities with a focus on CS3733 Software Engineering and CS4241 Webware: Computational Techniques for Networked Information Systems.

In order to create a simple yet robust method of communication, the team first settled on the HTTP subset of TCP/IP. This approach allowed the use of a standardized message protocol and greatly increased accessibility for web applications.

To further standardize communication between the core and client(s), we partly enforce and strongly encourage the use of a RESTful or REST-like API. REST stands for Representational State Transfer. The term was coined in 2000 by Roy Fielding. (Fielding, 2000) The basic idea of a REST architecture is that clients communicate with the server by presenting all the information that they need to fulfill their request, and that most of these requests center around the transfer of a resource. A RESTful API allows for cleaner URL strings, more complete client-server separation, and less complexity in student produced code.

The core application enforces the REST architecture in several ways. The first is to map HTTP request methods such as GET and PUT to data manipulation actions such as retrieving and saving. The basic core interface or the API maps these requests to similarly named methods in the EntityManager interface. The core framework also enforces clean URL strings by not utilizing query strings, instead encouraging JSON messages.

In addition to utilizing a REST architecture for the API, the core also makes use of JavaScript Object Notation (JSON) to encode and transmit content. JSON is a human readable way of encoding complicated objects into ASCII strings. The use of JSON in this context reinforces the client agnostic approach of the core framework. The core framework was developed in Java, but because all communication is done through the API, and this API uses JSON to encode objects returned to the client, the client can be written in virtually any language. In addition JSON is natively supported by most web browsers and there are JSON libraries available for most languages. (“JSON in Javascript”, n.d.)

3.5.5 SECURITY CONCERNS

In securing WPISuite TNG’s Core Server, we sought to achieve authentication and authorization across the system. The primary areas of concern for security in the project were the REST API and the webAdmin. Controlling access to these tools is important because they are the interfaces for outside users to manipulate the system. Further, access control and authentication offer developers more tools for their modules, like users and permissions.

To achieve these security goals, we implemented login authentication and session management services in the Server. These services are given to developers out-of-the-box such that

authentication is as easy as creating a user. Authorization is enabled by adding permissions to a Model.

Login Authentication

The login authentication service is contained within the WPILoginServlet class and the wpisuitetng.authentication package. The WPILoginServlet handles the HTTP requests for login and logout. These requests are passed on to the classes in the authentication package, where the authentication and password cryptography takes place. Figure 7 below depicts the class structure in the login authentication service.

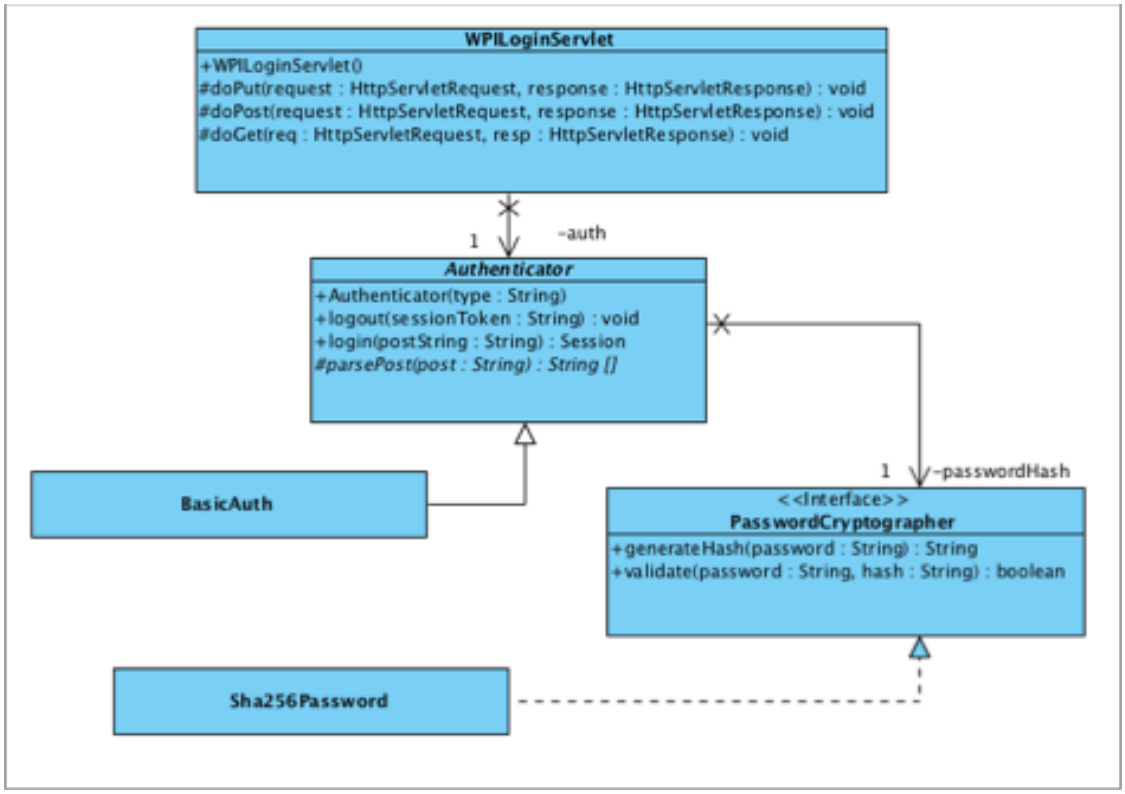

Figure 7 – WPISuite TNG Login Service Design

Authentication and password cryptography have both been abstracted behind an abstract class (*Authenticator*) and an interface (*PasswordCryptographer*), respectively. *Authenticator* handles authentication in the *login()* function by first parsing the user's credential pair (username/password) in the *parsePost()* function. The password section of the credential pair is then hashed by the *PasswordCryptographer* and matched against the user in the credentials.

Implementations of *Authenticator* are responsible for parsing user credentials from an authentication token provided in the login request's POST body. We chose a BasicAuth implementation because it is fairly universal and a straightforward token format. For our implementation of *PasswordCryptographer*, we chose the SHA-256 hashing algorithm.

Sessions and Session Management

After authentication, the SessionManager assigns a session to the newly authenticated and returns a session key to the user. For subsequent API calls, the user must include the session key with their request. The API uses the session key to authenticate the user requesting an entity. If permissions are enabled for the requested entity, then the API will check the user against the entity permissions. In this way, we provide authorization as well as authentication.

Each session contains a reference to the user, a timestamp of the login time, and a reference to the project the user has logged into. It also holds an instance of the session key as a randomly generated, string encoded, 64-bit signed long.

Upon logout, the user must provide their session key so that the SessionManager can remove the user's session from the pool.

3.5.6 ROBUSTNESS

When developing the WPISuite TNG's core systems, we wanted there to be a quick and simple process for students to be able to add new modules and functionality. With such a design in place, developed modules could have a long life span and start up very quickly. This mentality led us to the interface that became the EntityManager. The EntityManager interface dictates which methods the student modules would need to develop for interaction with their Data Models.

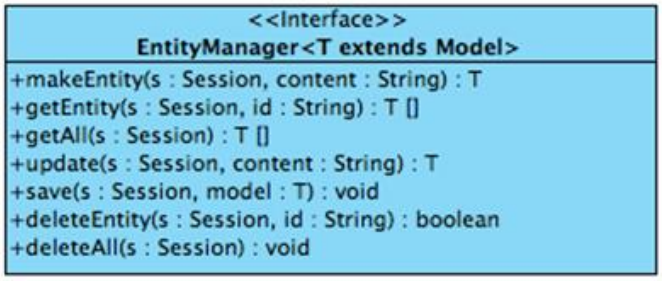

Figure 8 – WPISuite TNG EntityManager Interface

This interface outlines the seven methods required in order to develop a fully functioning model. We determined that seven would be the minimal number in order to fully satisfy the needs of every module. The seven methods can be broken down in how they interact with data entities with one method to create (makeEntity), one to retrieve (getEntity), one to update (update), one to save (save), one to remove (deleteEntity), one to retrieve all (getAll), and one to remove all (deleteAll).

Additionally in order to increase accessibility to students, a series of descriptive errors were created to aid in the troubleshooting process. Overall, the core can deliver 10 different specific errors in addition to 1 generic error. The WPISuite Exception is the generic exception in which ideally a helpful error string is returned to the user. Additionally, more specific errors can be

returned to the user relating to authentication, bad request formation, data conflicts, database exceptions, forbidden access, data not found, method not implemented, serialization, user sessions, and authorization.

Finally we implemented server side logging detailing all of the requests going into the server and what data is being returned from the server to aid in locating any errors that could have occurred in the actual transmission of data.

4 RESULTS

To determine the effectiveness of the project, we deployed the system to the students in Professor Pollice's Software Engineering class. They were tasked with building a requirements manager module. In the five weeks that the students had to work thus far they have made over 3,200 commits and added over 42,000 lines of code. Many of the teams have created effective requirements manager modules and one team even made a simple web application with the use of the core API.

However, we did not want to judge project success based on student team code metrics alone, we also wanted to see how the students felt about the system when they were using it. We sent out a survey to the students asking them questions about their setup process, the documentation, and their confidence in module building in various aspects. We used both the overall survey data and pivot tables and charts to be able to extract as much information from the 30 responses to the survey that were received.

In looking at the total survey information we found three things of particular interest. The first is summarized in the graph below.

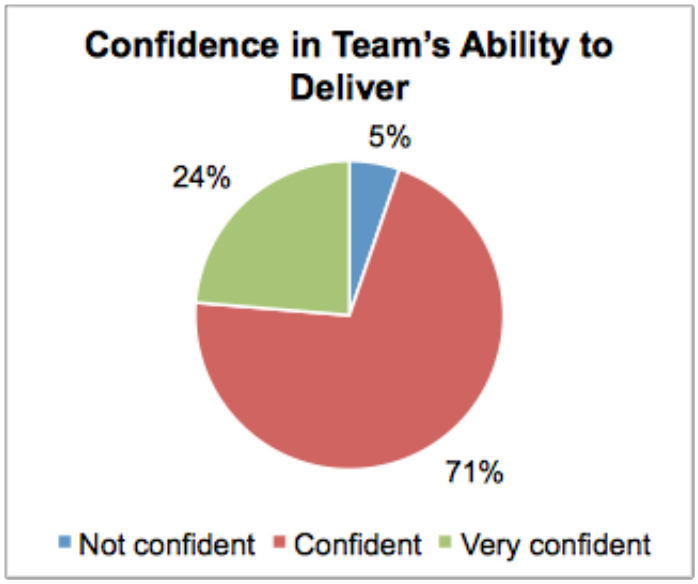

Figure 9 "Confidence in Team's Ability to Deliver" – Pie chart

Of the students that responded to the survey, we found that 95% of them were confident to very confident in their team's ability to deliver a robust and working module. The students are not only committing many times and adding lines of code to the project, but they are confident in their ability to produce something useful that can be used for years to come.

The second point of interest concerned the documentation of the system, summarized in the graph below.

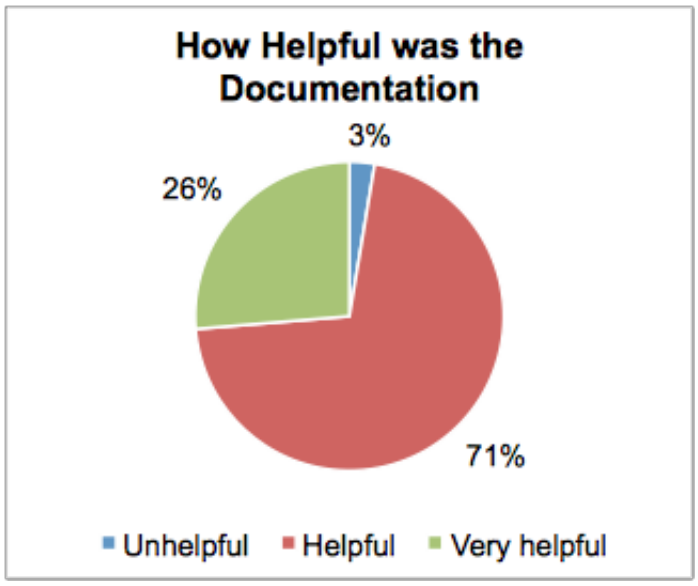

Figure 10 "How Helpful was the Documentation" – Pie chart

71% of the students found the documentation helpful, 26% found it very helpful and only 3% of the responders found it not helpful. The students found the documentation useful to them while learning the code base so that work could be started on the requirements manager module.

Last we considered the ease of setup for the system, pictured in the graph below.

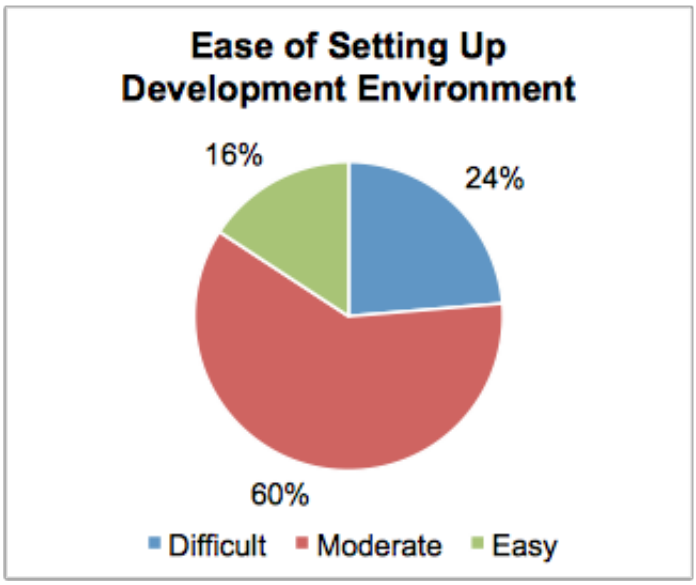

Figure 11 "Ease of Setting Up Development Environment" – Pie chart

76% of students found setting up the development environment to be moderately to easy difficulty. This was the area that proved the most difficult for the students who responded to the survey, but even at that only 24% of the responders found it difficult.

As well as the direct survey information, we garnered some additional information using pivot charts to separate the responders into categories and see if there were differences in their responses based on those categories. From this analysis we found two pieces of interest. The first focused on the ease of setting up the development environment based on the majors in the class, pictured in the graph below.

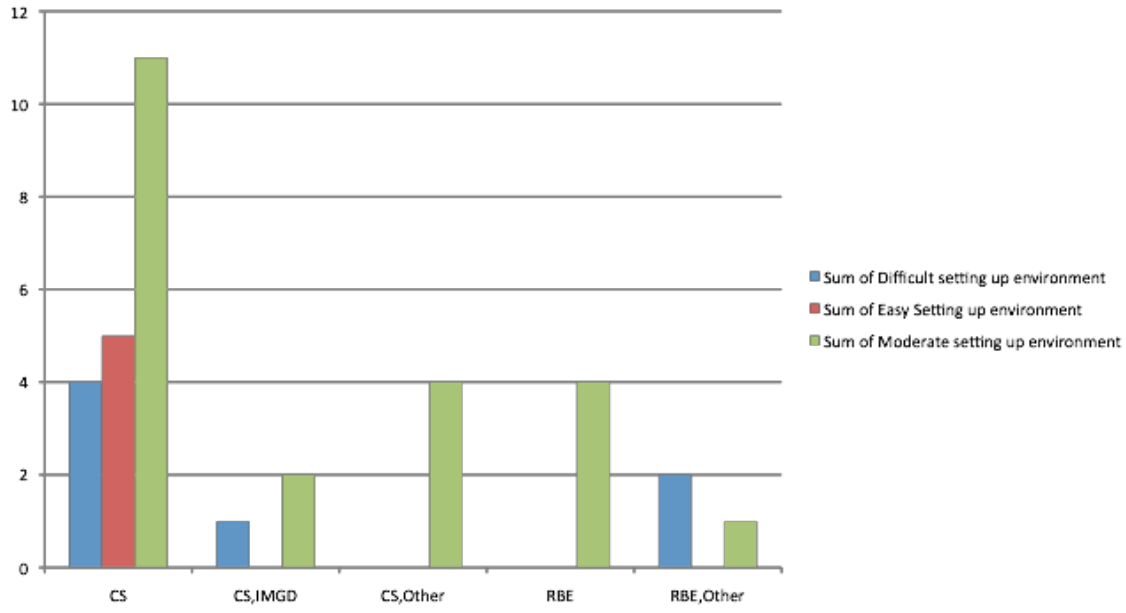

Figure 12 "Ease of Setting Up Development Environment" – Bar chart

While the distributions for most of the majors are fairly similar, the students who are pursuing the sole major of computer science had a much more difficult time setting up the development environment than their peers in other majors. 22% of the sole computer science majors had a difficult time setting up their development environment. Other majors did not have any responses saying that it was difficult to set up the environment.

The second focuses on the confidence of students creating their own module based on whether or not they had taken CS 3431, Database Systems I prior to taking the software engineering course, pictured in the graph below.

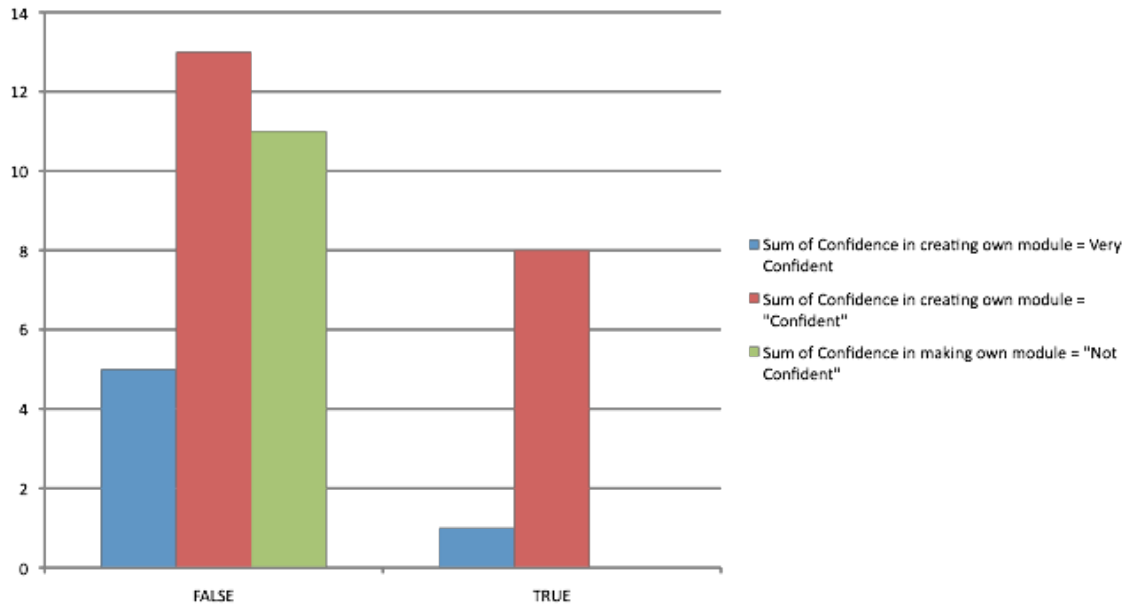

Figure 13 "Confidence in creating own module" – Bar chart

Of the students that responded to the survey, the students that had taken Database Systems course prior to Software engineering had a much greater level of confidence compared to those who had not taken the course. This is rather confusing because the Database Systems course focuses on SQL databases and the schemas required to create them, while WPISuite uses neither a SQL database or a schema because of the use of db4o, an object database.

5 LOOKING FORWARD

Moving forward with the product that has been developed, there are some areas that could be improved further. While none of them are critical to the functionality of WPISuite TNG, improving the security, creating dynamic module loading and setting a static location for the database file would make WPISuite TNG easier to use, and increased test coverage to prevent regression bugs.

The security could be improved by salting the passwords so that they would be more difficult to decode. Currently, someone working on the system is able to see the hashed password of any user they choose. With some work and the use of rainbow tables, the person would be able to figure out the password and gain access to the user's account. Rainbow tables are a method of decryption that reverses cryptographic hashing functions with the start point being the easily recoverable plaintext password. While deciphering the password still requires significant time, effort, and knowledge, salting the passwords will stop this common tactic from being able to break the encryption of the system.

Dynamic module loading is an aspect that would increase the convenience of the program. Currently, to load a new module or switch to a different module, it is required that the program is closed and reopened. By instituting dynamic module loading, this process would be able to be done dynamically allowing for new modules to be deployed while the program was still running.

An annoyance when developing with WPISuite TNG is the location of the database file. Since most development has been done through Eclipse, the location of the database file depends on how Eclipse launches the program. Depending on where Eclipse is launching it from, it can be in locations that are difficult to find, such as the package contents of Eclipse itself. Some of these locations also prohibit the use of db4o's Object Manager Enterprise (OME) that allows the user to view the contents of the database. By creating a static location for this file, it will allow the use of the OME no matter how Eclipse launches WPISuite and allow it to be located much easier.

Regression bugs were one of the biggest problems with the old WPISuite, when things were added or changed it was unknown if that change would break other components of the system. We have created many unit tests to help to make sure that regression bugs do not pop up, but because of the limitations of the test coverage tools we are not at 100% and are not sure if all of our code is covered by regression tests. By having an increased percentage of unit tests and a tool that can cover all kinds of unit tests, we can ensure that regression bugs are recognized immediately.

6 REFERENCES

Frey, R. (Designer). (2010). The model, view, and controller (MVC) pattern relative to the user. [Web Drawing]. Retrieved from http://commons.wikimedia.org/wiki/File:MVC-Process.png

(2006). Presentation-Abstraction-Control diagram [Web Graphic]. Retrieved from http://commons.wikimedia.org/wiki/File:Pac-schema.png

Cardenas, D. (Designer). (2012). Model-View-Presenter [Print Photo]. Retrieved from http://en.wikipedia.org/wiki/File:Model_View_Presenter.png

jBoss Overview. (2013). Retrieved from http://www.jboss.org/overview

Java Glassfish. (2013). Retrieved from http://glassfish.java.net/

Apache Geronimo. (2013). Retrieved from http://geronimo.apache.org/

Massè, M. (2012). *REST API Design Rulebook.* Sebastopal, California: O'Reilly Media, Inc.

Buschmann, F., et al. (1996). Pattern-Oriented Software Architecture Volume 1: A System of Patterns. England: Bookcraft.

Fielding, R. (2000). *Roy Fielding Dissertation: Chapter 5*. Retrieved from http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

JSON in Javascript. (n.d.). Retrieved from http://www.json.org/js.html