**SOFTWARE DEVELOPMENT REPORT – Jordan McDonald**

**I. SYSTEM SPECIFICATION**

In order to realize the dissertation research requirements it became critical to develop a software ‘workbench’ which can be leveraged in order to extract, visualize and analyze data extracted from the GitHub API. To do this an initial investigation had to be performed in order to determine the functionality that would be required and what can be achieved utilizing the data from the API. Section 1.1 outline each core system requirement the workbench will have to implement, sections 1.2 outlines constraints the system will encounter and

**A. CORE FUNCTIONALITY/REQUIREMENTS**

Data Extraction

Darwin.githubModule.send : function (url, callback, index, action)

darwin.AjaxResponseModule.performSuccessAction(action, response, callback, index)

**Description** - The GitHub API can accessed via HTTP ‘Get’ requests over the web with all data sent and received in the JSON data interchange format. To collect a specific metric from the API manipulation of the request URL is required, this typically follows this format ‘https://api.github.com/repos/projectOwner/projectName/metricType’ with each placeholder value replaced with those the user is targeting. In addition to this query parameters can be appended to the URL to modify the volume of data in the response, what ‘page’ of data will be returned (The API does not return the full dataset at once and has to be polled each page in turn) and user authentication details. The software system should allow the user to enter the GitHub projects URL and then A) collect the full dataset automatically B) allow the user to select the data they want to collect. The workbench will also be expected to iterate through each ‘page’ for a metric and merge the separate JSON responses into a single structure to ensure ease of processing in the latter stages.

Data Parsing

darwin.genericExtractorModule.extract(json, index, action, supplementData)

darwin.genericExtractorModule.getDateRange(lastKnownDate)

darwin.genericExtractorModule.addSamplesSkipped(currDate, lastKnownDate)

**Description** - Once the data is collected it becomes crucial to extract the data from the response that is required for the analysis, while discarding the unessential components and then organizing each metric for each project into a structured format. All data will be organized into a series of time vectors (at different sample rates – 1, 6, 13 & 26 weeks) and counts which represent that amount of times a specific metric occurred in any given time interval. Upon investigation it was discovered that data extracted from the GitHub statistics API (allows fast access to data used by the GitHub team itself) already formats the data in this way and minimizing the amount of requests required. However in comparison the majority of the metrics are not supported from the statistics API and return the raw unparsed data for metric instance, this reveals the need to develop an algorithm which iterates each dataset and performs an ordering process (based on the date) and organizes the data into counts based on a time interval while ensuring accuracy. Ideally this extraction process should remain generic and operate effectively invariant of the data type.

Data Storage

storeGenericAction.execute(request, response)

commitsDao.storeCommits(commits, database) \*and all the CRUD variants

**Description** - A crucial piece of functionality will be the ability to store the parsed data in an offline database in order to enable analysis of a large dataset while also saving the data when the application is not being utilized directly by the user. To reinforce the essential nature of this component we should consider the capabilities of the browser (the intended application will be web based) which cannot store vast volumes of data without crashing due to memory constraints so the data will have to be stored externally.

Data visualization

darwin.genericVisualiser.draw(values, sampleIndex, action, projectNames)

darwin.lawVisualiser.drawLawOne(values, projects) \*and all variants

**Description** - The workbench should have the ability to not only attain and parse data but also to visualize the information effectively in the form of graphs of charts. These should be generated on demand based on user input in some cases where in others should assemble automatically based on the full dataset stored within the database. Displaying the data in a visual manner is also an important feedback mechanism for the user in order to interpret and make inferences that isn’t possible when utilizing the raw parsed data, statistical data will also have to presented to account for the results required for the dissertation which is a key driver in deciding what features are essential.

Data Analysis

statsAction.execute(request, response)

lawsAction.execute(request, response)

**Description** - In order to fulfill the requirements of the dissertation the software system should enable statistical analysis that leverages the parsed data and generate results. The statistical measure should be varied enough to encompass the hypotheses that have been devised for the research dissertation while also supporting additional functions that will prove useful to any generic user. The goal of the system will be to not only support the dissertation but standalone as a robust application that can be used in future for further study.

\*See appendix A for the complete function definitions

**B. CONSTRAINTS**

Upon investigation it became clear that the system would be subject to a series of constraints partially as a consequence of the GitHub platform’s API and the fundamental nature of the problem itself. Each of these considerations will be listed below along with details on how the effect of each can minimized while maintaining the integrity of the systems core functionality.

* Data Extraction Time – if the user attempts to generate a large dataset using a significant volume of projects it may take hours to attain all the data required, this is mostly a side effect of the data mining process and is generally an acceptable expectation. When the target end point of a request is not encompassed in the statistics API then the maximum number of elements per response is only one hundred, and due to API restrictions this cannot be overcome.
* Rate limit – The initial amount of requests an unauthenticated user can make per hour is restricted to sixty however with application authentication this can be increased to five thousand which is enough to support an automated process. Despite overcoming this issue it should be noted that this is a viable blocker if the amount of request per hour exceed this threshold.
* API dependency – If the API in future is modified to such an extent where certain end points are no longer active it is possible the workbench will not function correctly with certain metrics. This is an acceptable constraint which raises the need to monitor the API for future changes, which in theory should be as simple as modifying the request URL

**II. SYSTEM DESIGN**

This section will cover each facet of the software systems design ranging from the user interface to

formally describing the interactions between the varying components of the system using the unified modelling language. The diagrams in this section represent an evolving process which began from initial UI sketches and class interactions to fully fledged representations of the system reflecting the need to adapt as requirements are refined based on unforeseen restrictions and also the reveal of novel aspects of the libraries selected that were identified. It should be noted that each of the designs reflect the system specification and bridges the gap between a description of the problem to an implementation in code ensuring a smooth transition from prototyping the system to refining the functionality to match the design documents while providing a stable reference point that ensures a well-conceived plan was followed, and in this particular case the system directly reflects the impact of good planning.

**A. USER INTERFACE DESIGN - STORYBOARDS**

Each UI component of the system was directly mapped to storyboards, in this case each web page within the web application corresponds to a single storyboard. Annotations and labelling has been provided which describes the expected functionality that is attached to each UI element while also describing the general purpose of the webpage in question. Each storyboard was designed using the drawing functionality contained within Microsoft Word which enabled satisfactory representation of each core component of the system and the expected layout – each blue corresponds to annotation which describing its functionality in this document.

Login Page Storyboard & Annotations



Social Media Icons

Register

Username Field

Password Field

Submit Button

Social Media Login

Login

This page represents the login screen which is the initial ‘splash page’ of the application, to access the system the user has different options to choose from. In this particular case the current state of this page reflects the selection of the ‘login’ tab (indicated in a red outline) which is similar to the register tab (which just asks for input confirmation) so it would be redundant to draw that alternative state. Each functional component of this web page has been labelled with a number, refer to the below list when describes the role of each labelled element.

1. In order to give the system a presence on social media it is prudent to give options to allow the user to ‘share’ or ‘like’ the system if they are using their social media account for utilizing the system. At this stage Facebook is the planned social media giant of choice but this may expand to account for additional domains.

2. The official logo for the system which serves no functional purpose aside from conforming to the ‘Darwin’ branding and heightening the visual aesthetic of the web page.

3. Each box represents a Bootstrap tab which is the main navigational element of the software, if register is selected the contents of elements enclosed within the tab structure will be updated to reflect the text fields required for registration.

4. Text fields which the user can utilize to input personal details required for the login process, the password field will be substituted with placeholder characters to ensure privacy.

5. The user has option to select social media external login or the custom Darwin process which will collect the data in the text fields, validate and store in MongoDB before redirecting the user to the main query web page.

Main Query Page

Database Options

Logout

Add URL

Visualizer

Statistics

Laws

Enter Repository URL Field

Get Data Button

Automate Data Collection

Input Repository

This webpage represents the main query page of the software, from here the user has two options – 1) input repository URLs manually for use in the manual graph/statistical process and 2) input a series of comma separated URLs and run the automated bulk data collection process. For a description of each labelled components functionality refer to the annotations below.

1. Database options that I planned to integrate include importing, exporting and wiping each collection – on selection these commands will be sent to the server which will perform the chosen action.

2. These buttons allow the user to access different functionality, each button will change the entre tab structure and substitute in its own version, for example if statistics is clicked the input repository tab will be removed and replaced with a series of statistical measure options.

3. The user can enter different repository URL into this field – the main point of custom input for the system.

4. This icon allows the user to dynamically generate new input fields if they wish to include more repositories as part of the manual process.

5. Once a URL has been entered and the submit button has been clicked, this icon will allow the user to get additional repository information by generating a pop up box.

6. These two buttons determine whether the automated process is activated or the manual process (the automated process gets the data all at once, while the manual process lets the user selected and visualize select metrics).

Typical Tab Structure

Database Options

Logout

Add URL

Visualizer

Statistics

Laws

Commits

Tags

Forks

Issues

…

Chart modifiers

Project selection

Chart/Table

The storyboard above shows the typical structure of a tab that resides in the statistics or visualization sections, the focal type is generally a chart or a table which presents information based on the user selection. This is a product of the manual process where the user independently selects the data and projects there wish to evaluate, each tab generally has variations in presentation but this is the template form which each is build, see below for annotations describing the core sections.

1. Each section will contain a series of navigational tabs to match other system input steps, changing the tab will replace the tab contents with those required for the selected tab.

2. These elements will allow the user to select a different type of chart, different time series sample and the projects which the user wants to generate results for/extract data from the API. Each selected project will be dynamically added to the chart/table and data extraction is only performed when the data does not already exists in the projects scope.

These designs formed the building blocks for each part of the UI, it was essential to keep a consistent layout and interaction process to ensure the system is simplistic to use and user friendly. The color scheme should also be considered, in design the plan was to keep the layout clean and this would be reflected in bold (blue) colors against a white background supplementing the look and feel of a robust system which is also easy to leverage

**B. DATABASE COLLECTION DESIGN (SCHEMA)**

It should be noted that MongoDB (my choice of database) does not enforce a strict document structure (for more information refer to section 3.2) however it is crucial at the planning stage to define the expected data composed in each document of a MongoDB collection. While Mongo can have varying fields within the same collection I have chosen to structure the dataset in a fixed manner to simplify the program source code and analysis logic. It should also be noted that MongoDB does not rely on a relational model so each collection should be interpreted as an independent entity, see figure 1 below for the planned collections and documents fields that are expected as part of this project.

|  |  |
| --- | --- |
| **Collection** | **Fields** |
| Users | Username (String), Password (String), Role(String) |
| Commits | Dates (Vector of Strings), Commits (Vector of ints), Project (String) |
| Contributions | Additions (Vector of ints), Deletions (Vector of ints), Difference (Vector of ints), LOC (Vector of ints), Dates (Vector of Strings), Project (String) |
| Correlations | Pearson (Double), Spearmann(Double), ProjectA (String), ProjectB (String), MetricA (String), MetricB (String) |
| Forks | Dates (Vector of Strings), Forks (Vector of ints), Project (String) |
| Growth Rate | Project (String), Metric (String), GrowthRate (Vector of doubles), OverallGrowth (double), Average growth (double) |
| Issues | Project (String), Dates (Vector of Strings), OpenIssues (Vector of ints), ClosedIssues (Vector of ints), AllIssues (Vector of ints) |
| Issues Comments | Project (String), Dates (Vector of Strings), Comments (Vector of ints) |
| Normality | Project (String), MetricType (String), Wilks (double), P-Value(double) |
| Mean | Project (String), Dates (Vector of Strings), Mean(double), MetricType (String) |
| Stars | Project (String), Dates (Vector of Strings), Stars (vector of ints) |
| Tags | Project (String), Dates (Vector of Strings), Tags (vector of ints) |
| Variance | Project (String), Type (String), Variance (double) |
| Cross Correlation | CrossCorr (double), Project (String), TypeA (String), TypeB (String) |

Figure 1 – Mongo collection data and datatypes schema

**C. UML DIAGRAMS**

To enable a transition from designing the system to actual development I decided to leverage UML diagrams in order to visualise the structure of entities and the sequence of actions a user can perform. This was performed utilising an eclipse plugin called ‘ObjectAid’ [4] which not only enables a drawing facility but model driven development from which code can be generated from a model, which will utimately improve maitenace and save time in certain cases.

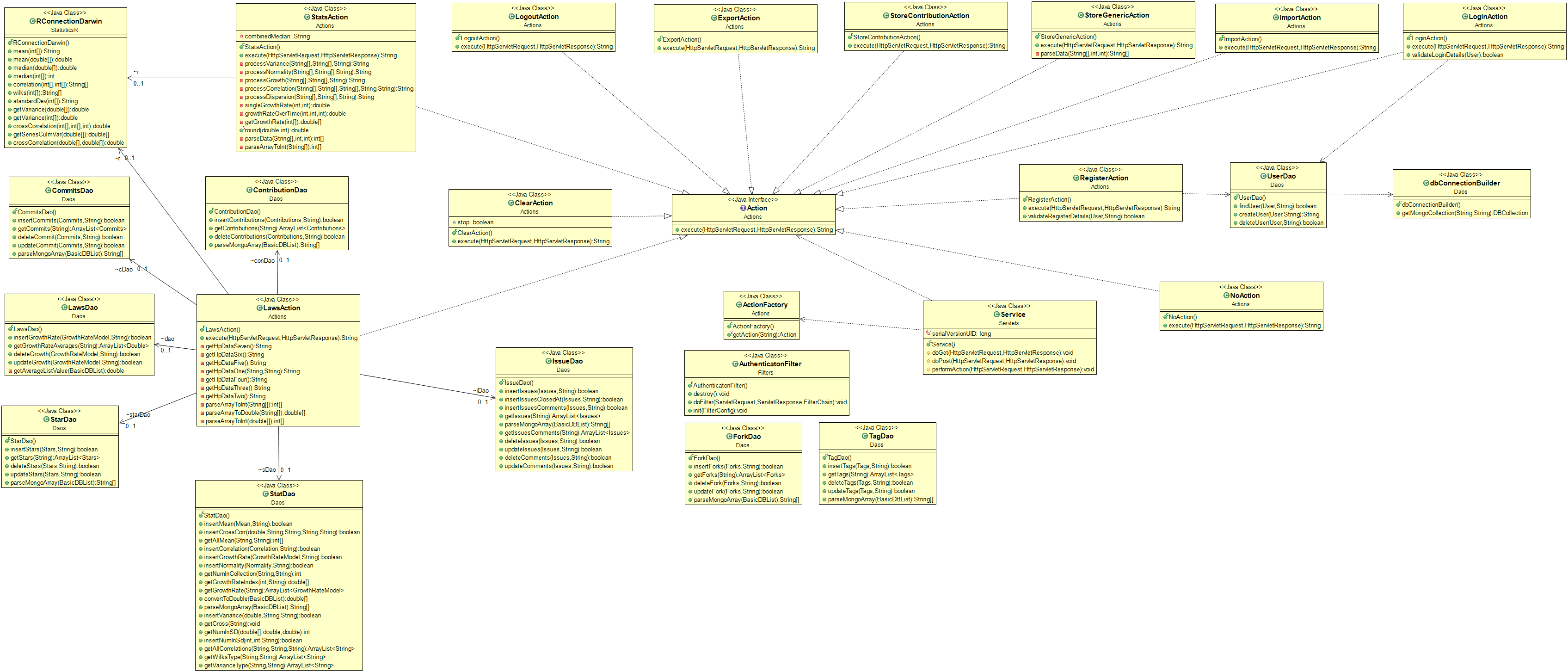
****

Figure 2 –class diagram showing the Java action classes

Figure 2 shows a class diagram highlighting the entities within the system that perform specific actions and the various processes that these leverage such as domain access objects. I am planning to utilize the factory design pattern (explain in more detail in section 3.3) which is reflected in this class diagram which show a series of action entities implementing the same common interface alongside the presence of an action factory class. The most complex class that is connected to the most entities is the ‘Law Action’ which uses the DAO classes (to get and insert data from the DB) and ‘RConnectionDarwin’ for statistical analysis, since the project revolves around providing metrics to visualize Lehman’s laws this is logical. To ensure the class diagrams remain easy to interpret the ‘model’ classes were moved into a separate diagram in figure 3 which shows how the data attained from the API was parsed into a ‘model’ and the classes which will use the models.

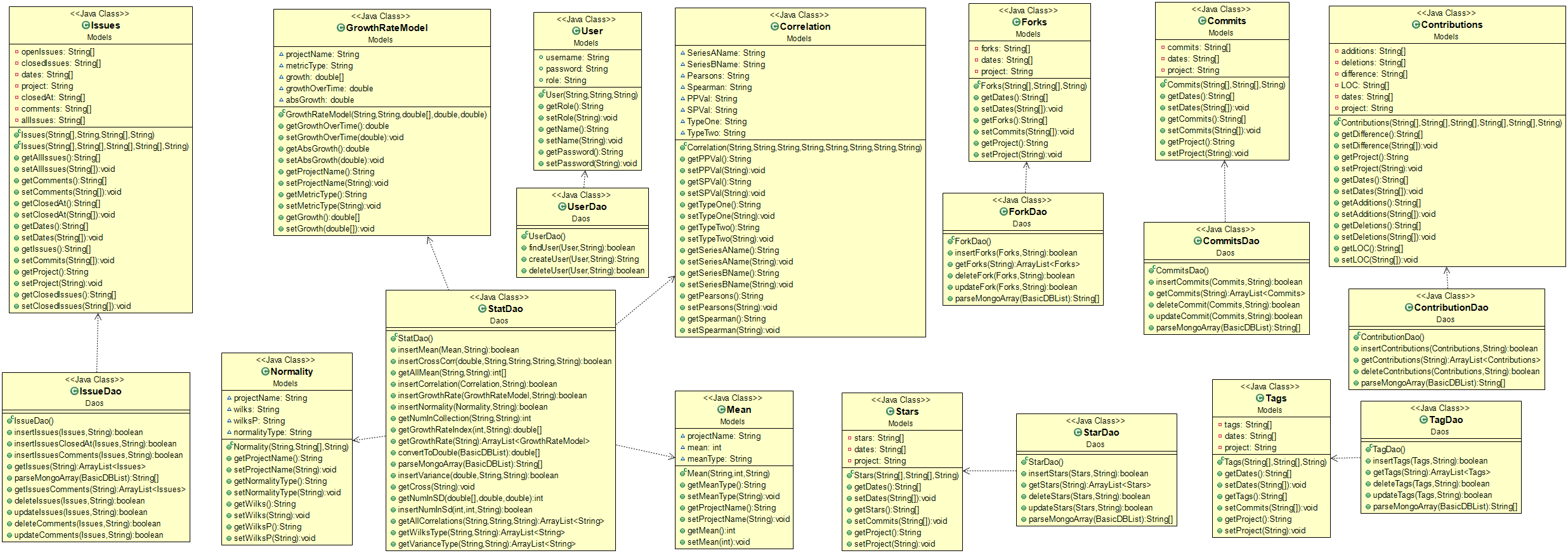


Figure 3 – class diagram showing the model classes

In this system each action that originates in the UI and ends in the server will typically follow a generic process to ensure ease of adding new features and maintainability. While each specific action performed will show deviation in the exact steps taken for the sake of this document (and minimizing the sheer volume of sequence diagrams that would be required to represent each user action) a generic process has been outlined in figure 4 which represents extraction and visualization steps which shows an end to end (UI to DB) sequence that the other actions will generally follow (albeit with different methods etc.) and provides a structure of interactions between different objects in the system.

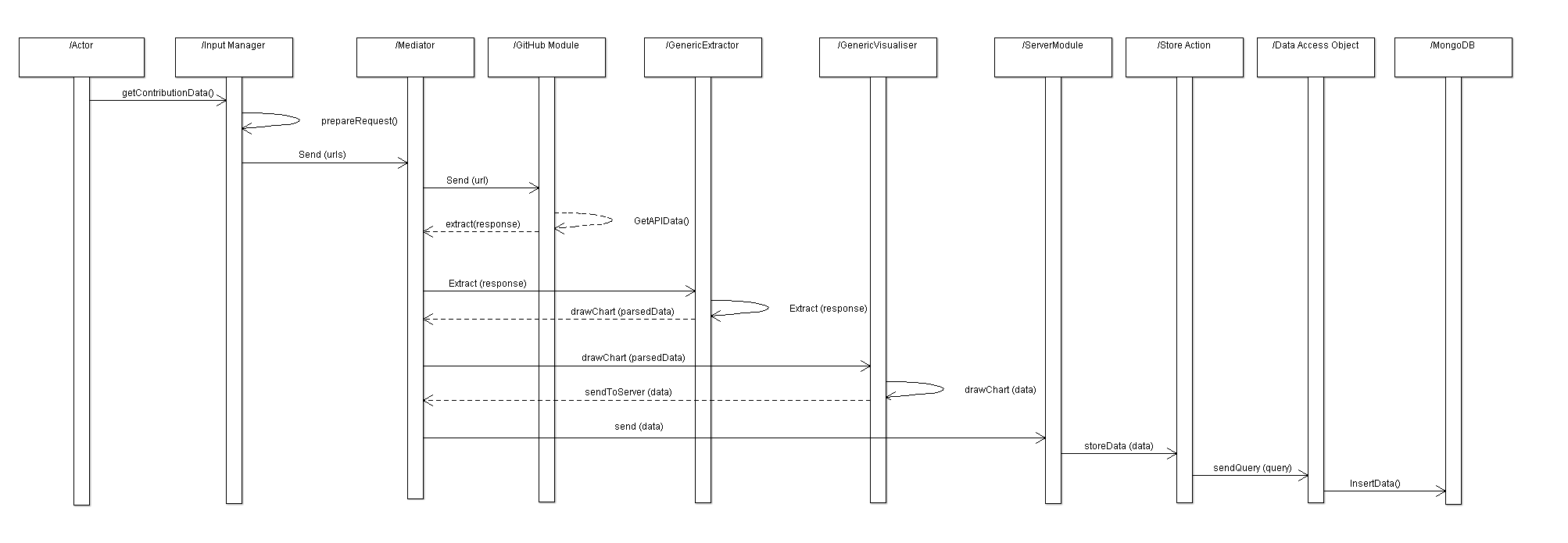


Figure 4 – sequence diagram showing a general approach to system interactions

**III. IMPLEMENTATION AND TESTING**

This section will examine in detail the tools, libraries and environments that have been leveraged in order to realize the system that has been planned in the previous sections of this report. A key facet of the implementation involved extensive research and planning in order to select an approach which meets the requirements of the project and enable the research that would facilitate the dissertation. Each of the key decisions made will now be examined in sequence initially discussing the programming languages selected and concluding with a description of the testing process and verifying the validity of the software system.

**A. PROGRAMMING LANGUAGES**

In order to meet the requirements of the project the context had to be considered, to form a dataset from the GitHub API it was crucial to select a programming language which enabled direct access to API via HTTP requests. To account for this JavaScript was selected and in tandem leveraged the JQuery library which simplifies HTML document traversal, DOM (document object model – a cross-platform and language-independent convention for representing and interacting with objects in HTML) manipulation, event handling and animation. However the main reason that drives this choice of library is the AJAX functionality which allows the webpage to dynamically send various requests to the API to account for user input or as part of automated process using callbacks. Additional reasons that support the selection of JQuery include JSON parsing and manipulation – the API will return data in the JSON data format (key value-pairs) and therefore it was crucial to have a system that could perform robust processing of this data, something JQuery enables. It should be noted that other options for interacting with the API were available, examples include Octokit (Ruby and C# versions) which is a GitHub endorsed alternative, however the project was not mature enough for this particular context and neglected the ability to make certain requests whereas through AJAX any functionality/data the API provides can be utilized.

Based on the advantages stated for selecting JavaScript a decision was made to frame the project as a web application, therefore consideration for libraries and tools that would bring the storyboard designs of the system to fruition. To fulfill this the Twitter Bootstrap JavaScript language was identified which provides a large selection of components to develop appealing user interfaces, in particular the ‘tab’ navigation would prove to be the focal point for all user interaction with the system. In addition to this Bootstrap provides a ‘mobile first’ approach which liquid displays that adjust to become aesthetically pleasing on different devices via pre compiled styling that would reduce the amount of micro-managing required by the developer. Visualization of the data extracted from the API is a key requirement that was partially facilitated by the use of the Google Chart library which can render various graphs in appealing ways, a dependency for this library was JQuery was solidifies that as a prudent initial decision. In order to provide additional variety to the presentation of data another component of JQuery was integrated, JQuery UI which provides additional options and was key in proving further ways to engage a user. In order to further accessibility and reduce load on the database the Facebook login SDK was utilized to externalize the user management process and offer and additional way to access the system. To see an example of JavaScript and these libraries working in tandem to create a vivid UI, see figure 5 below.

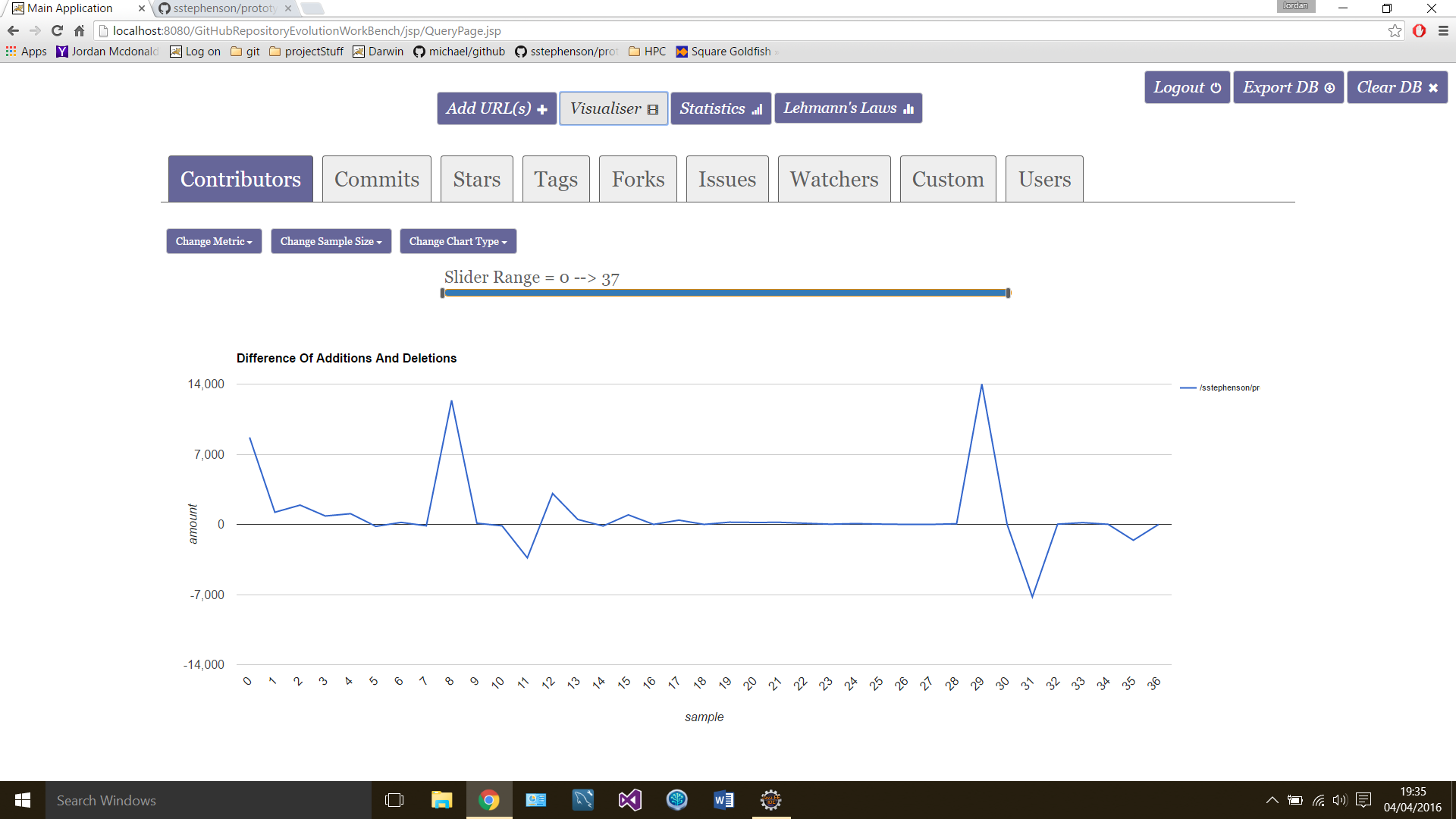


Figure 5 – an example of a completed UI tab

In order to Flesh out the capabilities of the web application to enable the storage and analysis of the mined data from the GitHub API a centralized server would be required, in this case the server side language is Java. This language provides an ideal method to interface with the web page via the Servlet technology which allows a server to send and receive HTTP requests in formats such as JSON so fits the overall architecture and work flow of the system being developed. In addition to this it was crucial that the server side language can communicate with the statistical analysis environment and the database technology (to be covered in the next section) which reinforces the choice of Java, as connectors and interfaces are provided which enable this process. To assist the default Java functionality a series of additional libraries were leveraged, GSON which performs serialization/deserialization of JSON to and from Java Objects, since all data instances on the server are modelled as Java beans this was a crucial library to standardize this process. In addition to this Reserve was utilized (a library to allow java to communicate with an R server instance) as well as Junit (for unit testing the system) and the Mongo Java driver in order to facilitate communication with the database, each of these will be examined in detail in the following section.

**B. SYSTEM ENVIRONMENT**

Initially the development environment has to be considered, for this project eclipse was selected as the IDE. Advantages of this choice include integrated configuration with the project files and the Apache Tomcat web server which will be utilized to host the web application and support the use of several JEE specifications to enable the servlets to send and receive requests. In addition to this Apache Maven can also be easily included into the work flow using eclipse and provides the option to automate the build process of the project, however at this stage has not been pursued but could become useful in future versions of the software, as well at this maven provides simplistic management of dependencies which was a key driver of using this functionality. Eclipse also provides different project configurations, in our scenario the ‘dynamic web project’ was selected which ideally prepares the system by generating a POM, Servlet dependencies and a consistent folder structure.

Now to discuss the various facets of the systems components that interact in order to provide the functionality in an efficient and cohesive manner that meets the requirements. Figure 6 shows a diagram that shows an overall view of each discrete part and the manner in which they communicate with each other. Two parts of the system are pending discussion, initially MongoDB will be considered which serves as the database for this application storing parsed API data and user details. MongoDB is a ‘no-SQL’ database which forgoes the typical relational model in place of documents that have dynamic schemas structured in a JSON format, each document is typically a member of a collection which contains a large number of similar documents – for example in a ‘commits’ collection will be a series of document each representing the commit data for one project. In this application context a database with no relations makes logical sense as each metric extracted from the API can be contained within a collection of that type and then composed of documents containing the actual data for a repository. Mongo also provides a driver to interface with Java, form this server side dynamic querying is possible in addition to exporting, resetting and dropping the database collections which add useful utility functionality to the project. In order to manage the database a GUI tool was utilized ‘Robomongo’ which connects to the mongo database and supports creating collections, removing documents and importing JSON documents and in general makes testing and managing a large dataset of documents more convenient.

In order to enable reliable and robust statistical analysis of the data the R software environment was chosen as the platform. Java can interface with R via Rserve which is a TCP/IP server which allows other programs to use facilities of R and provides a library of operations and datatypes to enable manipulation of the returned data. This statistical analysis is key in allowing the workbench to directly show results dynamically based on user input, whether the entire the database is evaluated to fulfill the research objectives or a statistical measure is performed on a select data from the UI.



Figure 6 – general system architecture

**C. DESIGN PATTERNS/IMPLEMENTATION TECHNIQUES**

In order to devise a system that meets the requirements and also follows best practices in regards to software engineering a series of design patterns where considered that would shape the low level approach of the coding process in order to enable ease of maintenance (e.g. fix bugs without impacting separate functionality) and adaptability (e.g. simplify adding new features).

The web specific portion of the application will be evaluated initially – the module pattern [1] was leveraged in order to encapsulate a discrete aspect of the systems functionality into a generic reusable ‘module’. Each module was contained within a global namespace (Darwin), benefits of this include minimizing the impact of external JavaScript libraries interfering with the custom written code (if variable names clash etc.) by designating a project scope, therefore to access any modules the namespace would have to be directly invoked which mimics the approach utilized by JS libraries. While JavaScript does not provide an ‘out of the box’ concept of privacy the module pattern can alleviate this by declaring variables and functions inside a module which prevents code from directly accessing the functionality without referencing the module itself which overcomes a key JavaScript shortcoming. However if it is desirable that a method if public the ‘return’ keyword can be utilized to give external module code access to the function/variable which provides flexibility in choosing what components have privacy enforced. In order to help visualize this process an example of a JavaScript module that is leveraged in the project is provided in figure 7 which highlights the concepts which have been examined.

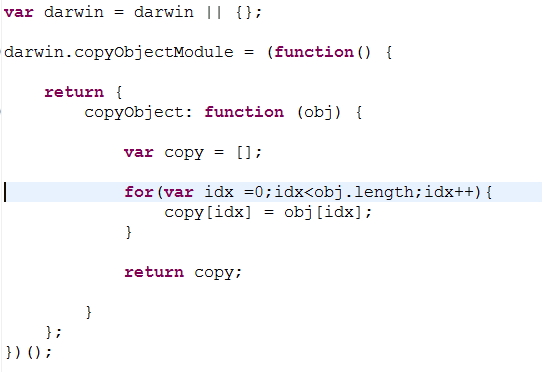


Figure 7 – example of a JS module

While the use of the module pattern has numerous advantages, it raises the question, how do the modules communicate with each other? A behavioral design pattern known as mediator was introduced. The mediator itself is represented as another JavaScript module with the purpose of coordinating data to and from other modules in the system which introduces low coupling as each module is an independent functional portion of code that do not rely on dependencies. The mediator was also given license to perform basic preprocessing to the data in some cases to ensure the module could operate as expected or to account for variation in structure of the different datasets extracted from the API. As the project increases in size beyond the current scope the Mediator may as a result become bloated, therefore it could become prudent to split the mediator into partitions to account for each facet of the system, its key to be aware of limitations to a design pattern as well as the strengths. A final consideration is even handling, each ‘tab’ of the system is bound to an ‘input manager’ which handles various user input and routes the selection (and data) to the Mediator for further processing. A general overview of the JavaScript program flow can be seen in figure 8.

User Input (UI)

Input Manager

Mediator

Module A

Module B

Figure 8 – demonstrating the typical JS design patterns leveraged

Design patterns implemented on the server include the creational factory pattern which handles the various types of ‘actions’ that need to be performed on the server ranging from handling the login process to storing data in MongoDB. Advantages of applying this pattern are include the ability to create a new object without exposing the creation logic to the client while executing using a common interface, in this case an ‘Action’ Java interface which is implemented by each specific action type to use an ‘execute’ method. In addition to this code adaptability and reuse is simplified as introducing a new class/modifying a class or action involves adding code to the factory rather than managing instantiation instance. The factory method has the responsibility of returning the correct action based on the input received in the HTTP request, the execute method is then called and the specific operation then proceeds as desired. See figure 9 for UML showing and example of this process.

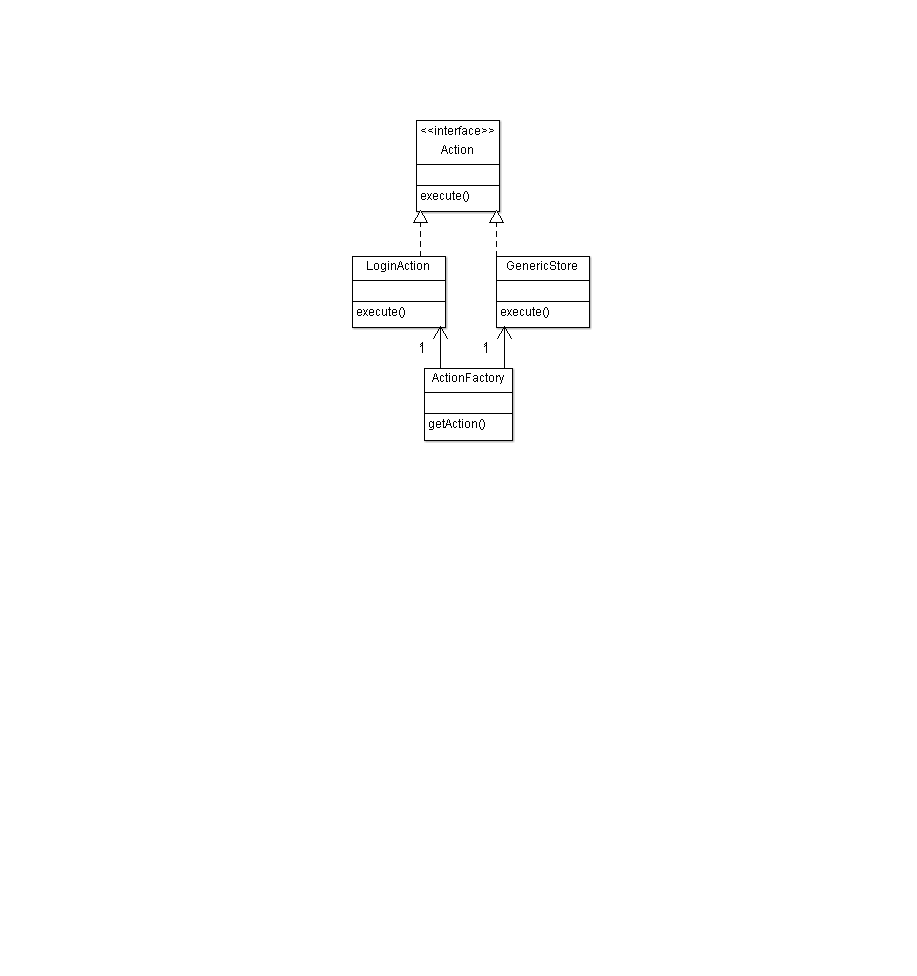


Figure 9 – factory pattern implementations

An additional best practice that was implemented was the use of data access objects (DAO). This technique enforces the single responsibility principle by designating all persistence (a different DAO for each Mongo collection) to the DAO which separates the application from the database to enable each to evolve independently without affecting each other, this heightens loose coupling and ensures all persistence logic for a collection can be accessed in one location. Unit test also benefits from this design pattern as the database operations can be tested in isolation effectively on a cloned test database without relying on application specific code (known as actions in this project).

**D. TEST STRATEGY/TOOLS**

A crucial component of any software system is not only development but ensuring that the system is robust, bug free and actually meets the requirements stated in the specification. Unit tests have been utilized in order to verify each ‘unit’ of the system (typically a unit equates to a class in OOP) by isolating it and applying data which represents each possible use case (failure handling, ‘normal’ data) to ensure the unit runs as expected. The advantage of unit testing is future changes can be automatically run against a test suite to determine errors/side effects have been introduced and furthermore unit tests require easily testable code which encourages the developer to partition source code into functions which meet this requirement. In the context of this project two unit testing tools where leveraged, to account for the lack of support for both JavaScript and Java so wielding specialist testing suites was chosen as the correct approach.

For JavaScript unit testing the QUnit library was chosen, this tool was developed by the team behind JQuery which reflects the cohesive nature of the selection of external libraries in the web based environment as each library stems off or utilizes the power of JQuery. In order to evaluate the results of a test assertions are performed which can perform checks such as ‘equal’, ‘deepEqual’ (for arrays) and even supports custom assertions when a particular case is not captured by the default functionality. To run the automated testing process the QUnit provided JS and CSS file should be included the header of a standalone HTML page along with any test files and files under test, then it becomes a matter of simply opening the HTML page and viewing the results – see figure 10 for an example run of the test used for this very project.



Figure 10 – the Qunit results window

In order to test the Java source code, Junit was selected, an established testing framework of choice for Java developers which has a presence on 30% of the Java projects out of 10,000 tested [2] making it the best most popular external library for Java projects. In addition to this Eclipse offers built in Junit visualization support (showing results) and can be easily added as a dependency via Maven and run within the IDE which reinforces the selection of this tool. A Junit test takes the form of a typical Java class which encapsulates a series of methods that are annotated with ‘@test’ in order to identify a test case, in some cases it will be required to setup the data used for the tests so the ‘@before’ annotation can be utilized to enforce ordering of method execution. Within each method Junit performs similar operation as we saw before in QUnit, we an assertion process of comparing an expected outcome against the result from the code under test. Within Eclipse all the test classes can be run at once and the results of an example Junit result can be seen in figure 11 below.

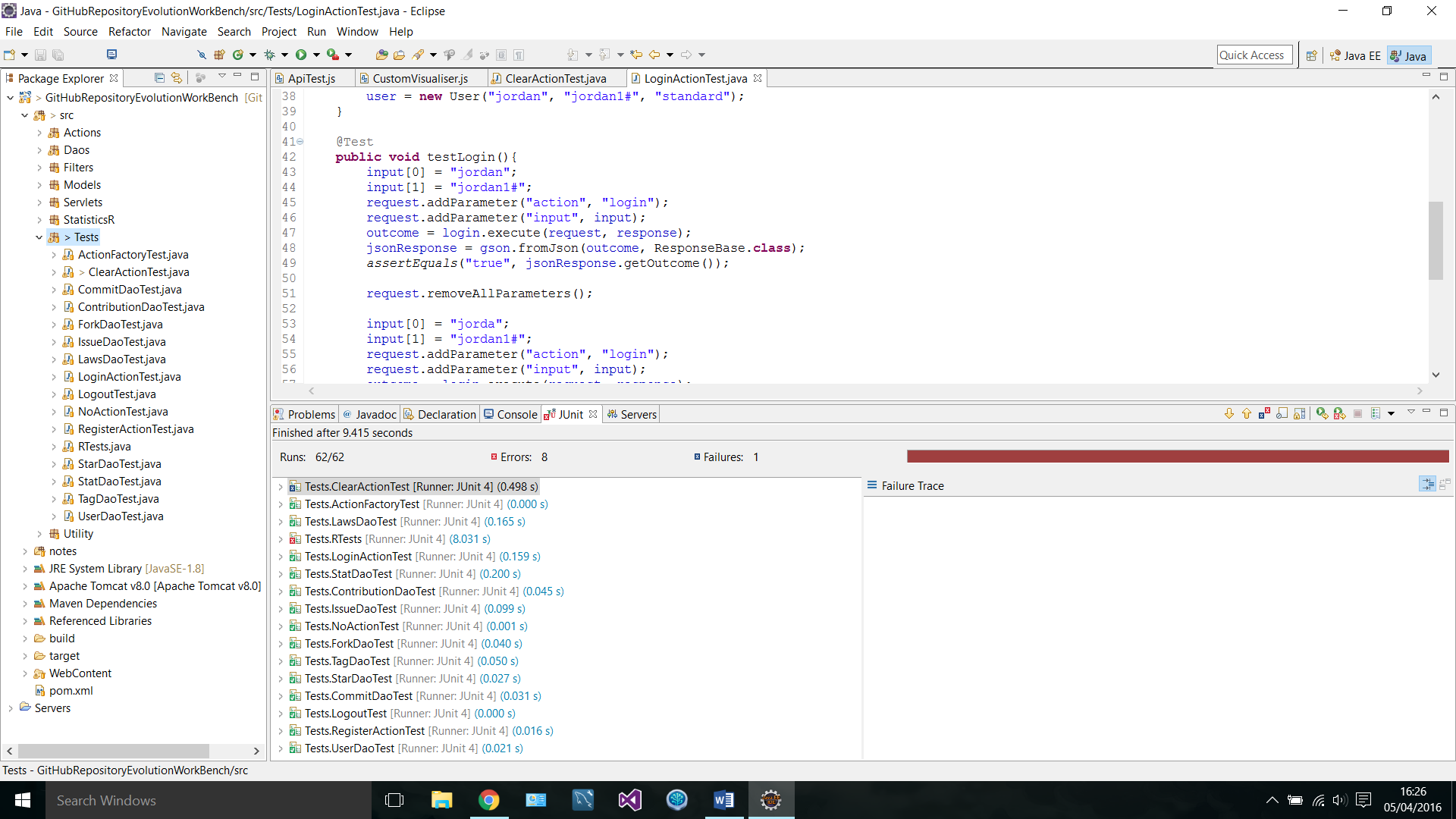


Figure 11 – example result of running the Junit tests

The general test strategy now needs to be considered, it was crucial to devise a standardized process of scrutinizing each test instance for a single unit. Initially the test will be performed using different variation of the type of data that is expected to enter the unit of code in order to determine it meets the requirements. Then the test strategy would evolve by introducing data that may introduce problems (empty array etc.), if the test failed on these instances then a cyclical process of adjusting to source code the account for these datasets became the next step as despite some extremes being unlikely it still improved the robustness and confidence in the software. In cases where it required data from the API the data was mocked using local variables to account for the fact the API data is constantly changing and cannot be relied upon for assertions. In order to test the DAO and MongoDB operations a test database was introduced to avoid polluting the main dataset, this follows a best practice in software engineering, each DAO is typically tested for the four main CRUD operations (create, read, update, delete) [3].

SYSTEM TESTING/MANUAL INTEGRATION TESTS

The main focus of this section rather than investigate how isolated components of the system perform (unit testing) a process of combining the different modules perform when combined utilizing real world data loads. Figure 12 shows the test performed and whether the outcome matches what is expected, various combinations of functionality have been considered in order to replicate a typical users interaction with the system.

**Test Setup** – five randomly selected projects of differing sizes

|  |  |
| --- | --- |
| **Test** | **Outcome?** |
| Evaluate the interaction between the contribution input managers, mediator, GitHub API, JSON manager, Contribution extractor & visualizer module but passing in the urls and clicking ‘get data’. | -Contribution data successfully obtained  -Data extracted in four time series intervals correctly  -Graphs render the data  -Graph options update UI as expected |
| Run the automated process on the projects that have been selected and verify if the data for each project is present in the mongo database. Test connection between UI, API, Server and Mongo while verifying the automated process. | -On monitoring the developer tools network tab the data for each project was collected correctly and in the correct sequence.  -The data collected was present within the MongoDB database. |
| Select ‘get data’ for the projects and then check is the modal for each is generated correctly. | -The modal for each project appears and disappears based on user input as expected and shows the correct data |
| Test the manual extraction of non-stat API data for each metric in sequence – verifies multiple requests types working as anticipated and the data for each metric filters through the system correctly. | -Each metric is obtained correctly in the manual process and visualized on each corresponding graph. The data is also present in the database. |
| Test each statistics tab functionality, verifies the R integration and the passes of the statistical data from the server to the UI for visualization. | -R calculates each statistic as expected  -the data is visualized correctly  -the data is stored in the database |
| Test the utility database functions but exporting, clear and importing the database in sequence. | -The data exports correctly to JSON  -The database collections clear as expected  -The data imports correctly |
| In sequence click on each law tab and determine if the data is generated correctly from R and returned from the server. | -Each laws tab successfully generates the data on the UI  -The various system components interact as expected |
| Test the ability of the server and UI to communicate via the login & registration process by first registering and logging in. | -The user data is stored in the database  -navigation to the main query page was successful. |

Figure 12 – shows a selection of integration tests performed

**E. IMPORTANT MODULES/FUNCTIONS**

This section will outline some core pieces of functionality that are critical to the operation of the software system. Each in turn has been described in the steps the algorithm/function loosely follows in order to achieve the required functionality, of course reviewing this description in tandem with the code will lead to a greater appreciation of these particular source code instances.

GitHub Communication Module – The send function of this module is fully generic and can make any possible request to the GitHub API. This is driven by the use of parameters which allow the URL to be passed in as argument, an index to represent the project and an action which identifies the metric that will be acquired from the API, however the main driver of this functions importance is the use of callbacks. Callbacks in JavaScript are typically function arguments that can then be executed with dynamic arguments (API JSON) when the Ajax call has succeeded, this allows the developer to route the response to any other function on the system using just this function. Prior to sending the ‘GET’ API request the URL has to be appended with a GitHub client id, secret id & access token which is attained from registering the application with GitHub, this gives access to an increased rate limit (the amount of requests per hour) from 100 to 5000 which is crucial when the automatic process is considered and the request volume sky rockets. Ajax in JQuery provides success and failure blocks which activate depending on the request status, if the request fails the feedback will be presented to the user and in the case of a successful request a callback will be made to process the response data.

URL

GITHUB API

PROJECT WORKBENCH

JSON DATA

Generic Visualizer Module – The purpose of this module is to draw a chart in any tab for any type of data (in our case time series organized counts). The process has different steps, initially an array of values (each value in the array is a vector) is assessed to determine the smallest inner vector to ensure the chart represents data that is available. Following this a google chart data table is initialized which contains values that will be shown on the chart and accepts values as either numbers or strings, using a loop each vectors values is added to the data table in sequence as different graph series. The options object is then prepared which utilizes function arguments to affect the rendering of the chart, following this the type of metric (and chart) is evaluated which determines the location the chart will be drawn an HTML id identifier.

Generic Extractor Module – The purpose of this module is to parse raw JSON data into four vector pairs, four vectors containing the counts for the metric at different sample points(1, 6, 13 & 26 weeks are the different sizes) and the associated dates. In certain metrics the data will need preprocessing, this could range from reversing the JSON to an ascending order structure or removing redundant data (removing pull requests from issues JSON). The next steps of the algorithm are driven by the date and forming counts based on the current weekly sample size, each element in the data is extracted using a loop which then increments a count if the associated data of the element is within the sample. This is achieved by calculating date range from the current date to the end of the sample on the first iteration (using the JavaScript date object operations), on subsequent iterations each new date instance is compared against the end of the sample to determine if it has been exceeded. In the case where it is within the sample the count is incremented, however when it exceeds the sample a calculation is performed to discover how many samples have been skipped (at times there is cases where long periods of inactivity are apparent) by polling different sample dates until one is found that contains the newest date instance, following this the array that holds the results own index is incremented to generate counts for a different time period. Once this process is complete the data is stored locally within JavaScript (this is turned off in large datasets – may cause a browser crash) and the data is transmitted to the server followed by visualization in the form of graphs.

GENERIC EXTRACTOR

RAW API JSON

1. DATE VECTOR AS WEEKLY INTERVALS

2. DATA VECTOR AS WEEKLY INTERVALS

Project Manager Module ‘handleAuto’ – Once the automated data extraction process has been selected the extractors will redirect the program flow to this function rather than visualization. The first step in this function is to reset all the ‘data managers’ and ‘JSON managers’ arrays to prevent browser crashing for a large dataset, following this the process begins by preparing the commit process for the first project in the list which then forms the commit dataset for that project via requests to the API. Once this step is complete each subsequent project in the list performs the same sequence until they all are populated fully, an if statement will detect is the current project is the last one and move onto a new metric using once again the first project. This cycle continues until the complete dataset is formed, while this function may not have the complexity of others its role in the system is paramount and this is reflected in its presence in this section.

**IV. APPENDIX**

**A. Function definitions**

**A1. Action Interface**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response);

Purpose – used to enable the factory design pattern, implemented by all factory classes.

Error Conditions – N/A

**A2. Action Factory class**

Signature – public static Action getAction(String input)

Purpose – based on the input the type of class instantiated will vary and return an ‘Action’

Error Conditions – if the action inputted does not exist/unexpected String – to handle this return a ‘noAction’ class with a message informing the user.

**A3. Clear action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – wipes the MONGO database collections

Error Conditions – N/A

**A4. Export action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – Exports the MONGO data to a folder specified

Error Conditions – if the folder file path does not exists a mesaage will inform the user to check the configuration

**A5. Import action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – Loads a folder of JSON files into the database

Error Conditions – if the folder file path does not exists a mesaage will inform the user to check the configuration

**A6. law action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – depending on the action sub type it redirect the program flow to account for a specific laws processing requirements.

Error Conditions – if the input identifier is incorrect – handled by returning a message to the user.

Signature – public String getHpDataOne (String type, String typeTwo)

Purpose – Perform the coordination with the data in the mongo database and R to allow statistical analysis of the raw data in order to answer the research hypothesis.

Error Conditions – Dependent on the functionaliy of R and mongoDB, if either fail the process will fail. Try-catch structures have been used to handle these occurences.

\*applies to getHpDataOne to getHpDataSeven to avoid repetition in this document.

**A7. login action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – Parses the request user data, performs validations and inserts the user into the session if they are successfully validated.

Error Conditions – failed validation – the user is shown a message informing them of this, from here they can try again.

Signature – public boolean validateLoginDetails(User user)

Purpose – Takes user details and performs validation, returning a Boolean as required.

Error Conditions – N/A

**A8. logout action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – removes the user from the session

Error Conditions – N/A

**A9. logout action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – removes the user from the session

Error Conditions – N/A

**A10. noAction action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – handles the case where the user has performed an action not supported by the user, returning a message.

Error Conditions – N/A

**A11. Register action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – takes user details and validates and then inserts into the database coordinating with the DAO.

Error Conditions – failed validation – returns a message to the user.

**A11. Stats action class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – depending on the input details, the program flow is redirected to other methods in the class.

Error Conditions – if the action is not supported, return a message to the user.

Signature – private String processDispersion(String[] data, String[] projects, String type)

Purpose – takes the input data and calculates the dispersion metrics. The results are then returned as a JSON string.

Error Conditions – dependent on R, try-catch constructs catch any instances where the calculation fails.

Signature – private String processCorrelation(String[] data, String[] projects, String type)

Purpose – takes the input data and calculates the correlation metrics. The results are then returned as a JSON string.

Error Conditions – dependent on R, try-catch constructs catch any instances where the calculation fails.

Signature – private String processNormality(String[] data, String[] projects, String type)

Purpose – takes the input data and calculates the normality metrics. The results are then returned as a JSON string.

Error Conditions – dependent on R, try-catch constructs catch any instances where the calculation fails.

Signature – private String processVariance(String[] data, String[] projects, String type)

Purpose – takes the input data and calculates the variance metrics. The results are then returned as a JSON string.

Error Conditions – dependent on R, try-catch constructs catch any instances where the calculation fails.

Signature – private String processGrowth(String[] data, String[] projects, String type)

Purpose – takes the input data and calculates the growth metrics. The results are then returned as a JSON string.

Error Conditions – dependent on R, try-catch constructs catch any instances where the calculation fails.

**A12. Store Generic Action Class**

Signature – public String execute(HttpServletRequest request, HttpServletResponse response)

Purpose – depending on the input details, the data will then be passed to a metric specific DAO will inserts the data to the database.

Error Conditions – if the action is not supported, return a message to the user.

**A13. All DAO Classes (same functionality, different metrics)**

Signature – public String insert(Metric metric, String database)

Purpose – inserts the data into a collection contained within the mongo database

Error Conditions – N/A

Signature – public String read(Metric metric, String database)

Purpose – reads data from the database based on the metric type

Error Conditions – If the collection is empty, return a message to the user.

Signature – public String update (Metric metric, String database)

Purpose – alter a document in a collection by providing new data

Error Conditions – if the pairs of documents do not match return a message to the user

Signature – public String delete(Metric metric, String database)

Purpose – delete a document from the database

Error Conditions – if the document does not exist inform the user.

**A14. All Model Classes (One for each metric type)**

Signature – public String[] getDates()

Purpose – gets an array of dates for the metric

Error Conditions – The array will be instantiated by default which can be handled by the system if no set has occurred.

Signature – public void setDates(String[] dates)

Purpose – sets the model array with the array provided

Error Conditions – N/A

Signature – public void setProject(String project)

Purpose – sets the model project name with the data provided

Error Conditions – N/A

Signature – public String getProject()

Purpose – returns the project name for this model instance

Error Conditions – N/A

Signature – public void setMetric(String[] metric)

Purpose – set the metric array

Error Conditions – N/A

Signature – public String getMetric()

Purpose – returns the metric array for this model instance

Error Conditions – N/A

**A16. Service Class**

Signature – public void doGet()

Purpose – passes the request and response objects to the action function

Error Conditions – N/A

Signature – public void doPost()

Purpose – passes the request and response objects to the action function

Error Conditions – N/A

Signature – public void performAction()

Purpose – interacts with the action factory to get the class required and executes the factory function.

Error Conditions – N/A

**A17. Service Class**

Signature – public void doGet()

Purpose – passes the request and response objects to the action function

Error Conditions – N/A

**A18. RConnection Darwin Class**

Signature – public String mean(int[] data)

Purpose – Calculates the mean using R for the given dataset

Error Conditions – if the datatset is empty then the processing is not performed and the user is informed.

Signature – public String correlation(int[] data, int[] dataTwo)

Purpose – Calculates the correlation using R for the given dataset

Error Conditions – if the datatset is empty then the processing is not performed and the user is informed.

Signature – public String normality (int[] data)

Purpose – Calculates the normality using R for the given dataset

Error Conditions – if the datatset is empty then the processing is not performed and the user is informed.

Signature – public String median (int[] data)

Purpose – Calculates the median using R for the given dataset

Error Conditions – if the datatset is empty then the processing is not performed and the user is informed.

Signature – public String getVariance (int[] data)

Purpose – Calculates the variance using R for the given dataset

Error Conditions – if the datatset is empty then the processing is not performed and the user is informed.

Signature – public String crossCorrelation(int[] data, int[] dataTwo)

Purpose – Calculates the cross correlation using R for the given dataset

Error Conditions – if the datatset is empty then the processing is not performed and the user is informed.

**A19. Ajax Response JS Module**

Signature – handleSuccess: function (action, response, callback, index, isTest)

Purpose – based on the action type and data form, sotres the data in the JSON manager and then a) starts the next request process b)detects the JSON is empty and calls the extractor

Error Conditions – All cases of the response format and arguments will be handled by null checks which can then inform the user.

Signature – getSizeOfArray : function(array)

Purpose – takes an array and returns the size, accounts for missing data values

Error Conditions – if the array is empty the processing is not performed

**A20. Array Utility JS Module**

Signature – getSmallestArray: function (array)

Purpose – in the case where an array conatins arrays, this returns the size of the smallest

Error Conditions – empty array checks are performed prior to processing at each index

Signature – getSmallestGenericArray: function (array, sampleIndex) {

Purpose – gets the smallest array within a specific index of a container array

Error Conditions – empty array checks are performed prior to processing at each index

Signature - trimArray : function(array, difference){

Purpose – reduces the size of an array by trimming off the end n amount of values

Error Conditions – empty array checks are performed prior to processing at each index

**A21. Contribution Extractor JS Module**

Signature – extract : function(json, index)

Purpose – takes raw json data and uses the date values to organize it into sampled weekly series as a vector which can then be analyzed.

Error Conditions – initial checks handle the case where any of the input values are incorrect

Signature – getIterationCount : function(seriesA, seriesB)

Purpose – takes two vectors as input and returns the shortest length

Error Conditions – N/A empty checks are been performed in the module already.

Signature – resetVariables : function() {

Purpose – each time new input is selected the variables are reset

Error Conditions – N/A

**A22. Copy Object JS Module**

Signature – copyObject: function (obj)

Purpose – JS is pass by reference with objects, to bypass this copy the object so it can be modified without changing the stored version.

Error Conditions – if the array is empty don’t perform the function contents

**A23. Custom Tab JS Module**

Signature – setupUiDropDown : function ()

Purpose – generates the dropdown list options based on the projects stored in the project manager module

Error Conditions – if there are no projects input, do not display anything

Signature – setupUiOptions : function()

Purpose – generates the check boxes for the metrics that have data avaliable

Error Conditions – some projects will have no data, in this case don’t show anything

Signature – clearComponents : function()

Purpose – removes the UI components from the page

Error Conditions – N/A

Signature – resetChecks : function()

Purpose – Resets a series of Booleans which represent what metric has been clicked

Error Conditions – N/A

Signature – setMetricChecked : function() \*one for each metric

Purpose – sets the metric checked status to true for future use when submit is clicked

Error Conditions – N/A

**A22. Data Manager JS Module**

Signature – setMetric: function (data, index)

Purpose – inserts the data into a specific index that is associated with each project.

Error Conditions – performs a length check (for 0)

Signature – getMetricIndex: function (index)

Purpose – gets a metrics data at a certain index

Error Conditions – N/A

Signature – getAllMetric: function ()

Purpose – gets the whole container array

Error Conditions – N/A

Signature – reserAllDataManager: function ()

Purpose – sets each metric array to the default form - empty

Error Conditions – N/A

\*each method above is implemented for each of the n amount of metrics.

**A23. Date Manager JS Module**

Signature – convertDateObjectToString : function(dates)

Purpose – converts a date JS object to a string for storage in mongo

Error Conditions – N/A

**A24. Facebook SDK JS Module**

Signature – darwin.statusChangeCallback = function(response)

Purpose – Loads the facebook SDK, displays the login option and handles the login process.

Error Conditions – Dependent on Facebook, if an error occurs with the SDK the functionality is not displayed to the user.

**A25. Generic Extractor JS Module**

Signature – extract: function (json, index, action, supplementData)

Purpose – takes the JSON data for each project and samples into weekly intervals based on the date associated with each response element.

Error Conditions – A series of checks initially determine if the data required is populated

Signature – getIterationCount : function(seriesA, seriesB)

Purpose – takes two vectors as input and returns the shortest length

Error Conditions – N/A empty checks are been performed in the module already.

Signature – resetVariables : function() {

Purpose – each time new input is selected the variables are reset

Error Conditions – N/A

**A25. Generic Visualiser JS Module**

Signature – draw : function(values, xAxis, title, sampleIndex, action, chartType, projectNames)

Purpose – Creates and renders a google graph based on the data input alongside the chart configuration arguments.

Error Conditions – Dependent on google charts library, if that changes the chart functionality may suffer

**A26. ISO601 to Date JS Module**

Signature – convert: function (dtstr)

Purpose – takes a date in ISO format and converts to a JS object

Error Conditions – N/A

Signature – getMonthLength : function(numMonth, year)

Purpose – gets the length of a month based on the year and month itself

Error Conditions – N/A

**A27. Json Manager JS Module**

Signature – getMetricJsonIndex:function(index)

Purpose – returns a metric based on the index

Error Conditions – N/A

Signature – getAllMetricJson:function()

Purpose – returns the full set of arrays for a metric

Error Conditions – N/A

Signature – resetAllData:function()

Purpose – resets the arrays to an empty form

Error Conditions – N/A

\*each function above is implemented for n amount of metrics

**A28. Law Visualiser JS Module**

Signature – drawLawOne : function(data)

Purpose – Updates the html page to account for the data retrieved from the server to answer the law

Error Conditions – N/A

Signature – drawLawOne : function(data)

Purpose – Updates the html page to account for the data retrieved from the server to answer the law

Error Conditions – N/A

Signature – drawLawTwo : function(data)

Purpose – Updates the html page to account for the data retrieved from the server to answer the law

Error Conditions – N/A

Signature – drawLawThree : function(data)

Purpose – Updates the html page to account for the data retrieved from the server to answer the law

Error Conditions – N/A

Signature – drawLawFour : function(data)

Purpose – Updates the html page to account for the data retrieved from the server to answer the law

Error Conditions – N/A

Signature – drawLawFive : function(data)

Purpose – Updates the html page to account for the data retrieved from the server to answer the law

Error Conditions – N/A

Signature – drawLawSix: function(data)

Purpose – Updates the html page to account for the data retrieved from the server to answer the law

Error Conditions – N/A

Signature – drawLawSeven: function(data)

Purpose – Updates the html page to account for the data retrieved from the server to answer the law

Error Conditions – N/A

**A29. Load Graph JS Module**

Signature – load: function ()

Purpose – loads the chart component of the google graph library

Error Conditions – runs on the assumption the google chart functionality will remain invariant

**A30. Modal visualiser JS Module**

Signature – drawModal: function (title, body)

Purpose – displays a model on the page based on the input data

Error Conditions – runs on the assumption the google chart functionality will remain invariant

**A30. Parse URL JS Module**

Signature – parse: function (url)

Purpose – Takes a url and extracts components of the path required to identify the project

Error Conditions – Assumes the URL input is standardised

**A30. Progress Bar JS Module**

Signature – updateProgressBar: function ()

Purpose – Updates the HTML element in the UI as feedback to the user

Error Conditions – N/A

Signature – updateMetricProgress: function (value)

Purpose – Each metric has an associated counter showing the requests made, this is charged with updating that counter.

Error Conditions – N/A

Signature – resets: function (value)

Purpose – resets all the progress indicators to 0

Error Conditions – N/A

**A31. Server Communication JS Module**

Signature – send: function (action, callback, type, input)

Purpose – Sends the input data to the server alongside the action to be performed at the server.

Error Conditions – non responsive server – in this case a message is displayed to the user in the case of a 404 HTTP error

**A32. Stat Visualiser JS Module**

Signature – drawMean : function(values, projectNames, metricType, standardDev, means, medians, collatedMedian)

Purpose – draws all the mean statistics on the HTML page

Error Conditions – N/A

Signature – drawNormality : function(values, projectNames)

Purpose – draws all the normality statistics on the HTML page

Error Conditions – N/A

Signature – drawCorrelation : function(values, projectNames)

Purpose – draws all the correlation statistics on the HTML page

Error Conditions – N/A

Signature – drawGrowth : function(values, projectNames)

Purpose – draws all the growth statistics on the HTML page

Error Conditions – N/A

Signature – drawVariance : function(values, projectNames)

Purpose – draws all the variance statistics on the HTML page

Error Conditions – N/A