Team Selene

Software Testing Plan

Visualization of High Dimensionality Spatial Data

March 28, 2017

Project Sponsor: Dr Jay Laura, USGS Astrogeology

Faculty Mentor: Dr Palmer

Team Lead: Daniel Ohn

Team: Zowie Haugaard, Christopher Philabaum, Kelvin Rodriguez, Makayla Shepherd

1. Introduction

Team Selene is a group of undergraduate students at Northern Arizona University designing and developing a web application for the United States Geological Survey to improve the access and analysis of the Kaguya Lunar Orbiter's spectral profiler dataset. This dataset, which consists of some 68 million points with associated reflectance and metadata, is 1.4 terabytes in size, captures important spectral image data from the lunar surface at a broad spectral coverage from visible to near-infrared light spectrum. This hyperspectral reflectance data can be utilized by planetary scientists and geologists to study the composition of the lunar surface, as well as our moon's and even Earth's origins and formation. The current methods for accessing this dataset, mainly through the downloading of individual images from a online archive and manually plotting the data through a geographical information system (GIS), are time consuming, involve a high amount of manual overhead, and do not support in any way exploratory analysis of the data. In order to make the Kaguya Spectral Profiler dataset highly accessible and to improve the way in which scientists may interact with and explore the data, Team Selene is designing and implementing a novel web based application to deliver the spectral profiler dataset and to provide tools to allow for its analysis and exploration.

The purpose of this document is to outline Team Selene's plan for ensuring the functionality of our application, here forward referred to as the Kaguya Spectral Profiler Explorer (Kasper), satisfies the specific requirements of this system. This will be done through the testing of functional and nonfunctional requirements of the system, resulting in quantifiable results. This document will outline these tests, which are spread between three distinct sections: unit testing, which will describe the tests aimed at specific methods within our application's implementation; integration testing, which outlines the methods used to test the interfaces between the separate components of the application; and usability testing, which describes the testing used to ensure the end user's ability to properly interact with and access all necessary functionality of the system.

Unit testing of the system will focus primarily on two components of the Kasper application: the RESTful API used to serve specific point and hyperspectral data, and the client-side static application interface which both makes requests to the API and renders the results and provides the user interface for the application. Integration testing will test the application's major interfaces: that between the database and geoserver, which generates a spatial visualization of the dataset; between the client application and geoserver, which delivers the spatial visualization to the client; and between the client and the RESTful API, which delivers specific point and hyperspectral data. Finally, through usability testing we will determine quantitatively the user's ability to access and interact with the application through the client side front-end web application.

2. Unit Testing

2.1 RESTful API

The RESTful API service, which responds to requests from the client with geospatial point and hyperspectral data, will require unit testing of the services it provides. The two API services provided are GET requests for Points and Image, which return JSON data used to render visualizations in the client's browser. Unit testing of these services will be as follows.The tests will be carried out by use of assertions, specifically based on the assertion library Chai. These assertions will be tested using the javascript based Mocha testing framework for Node.js, and will be performed from the server.

2.1.1 GET Points

The GET Points service returns a JSON object containing an error code and an array of Image objects containing the point data of some 1500 observations. If the request fails, an error code and error message should be returned. The unit testing of this API service will be performed through the checking of these properties through assertions. Two tests will be performed: one simulating an API call using a http GET request, and the other will test the points method itself.

1. GET Points Test
    a. request.should.have.status('200')
    b. request.should.be.a('object')
    c. request.should.have.property('error').eql(0)
    d. request.should.have.property('Images').be.a('array')
    e. request.should.have.property('Images').len(1560)

2. Points test
    a. assert.isNotNull(points)
    b. assert.isNotNull(data)
    c. assert.isEqual(data.error, 0)
    d. assert.isArray(data.Images

2.1.2 GET Image

Like the Get Points service, the GET Image service should also return an array, however this array will contain a single JSON object holding the hyperspectral data associated with a specific point. Unlike the Points service, a GET Image request requires two parameters: an ID, which specifies the Image ID for the requested observation, and an index, which specifies the index of the requested point in the image observation list.

1. GET Test
    a. request.should.have.status('200')
    b. request.should.be.a('object')
    c. request.should.have.property('error').eql(0)
    d. request.should.have.property('Image').be.a('array')
    e. request.should.have.property('Image').len(1)

2. GET Index Out of Bounds Test
    a. request.should.have.status('200')

   b. request.should.be.a('object')
   c. request.should.have.property('error').eql(1)
   d. request.should.have.property('Image').be.a('string')
   e. request.should.have.property('Image').eql('No Image Found')

3. GET Invalid ID Test
   a. request.should.have.status('200')
   b. request.should.be.a('object')
   c. request.should.have.property('error').eql(1)
   d. request.should.have.property('Image').be.a('string')
   e. request.should.have.property('Image').eql('No Image Found')

4. Image Test
   a. assert.isNotNull(images)
   b. assert.isNotNull(data)
   c. assert.isEqual(data.error, 0)
   d. assert.isArray(data.Images)

2.2 Client Javascript Application

The Client Javascript Application is made up of two major components: a map generated with the mapping library leaflet used to render geospatial data and graphs created using the Chart.js visualization library for displaying hyperspectral data. The application contains several helper methods used to create these visualizations, but the high level of abstraction that the libraries being used afford has reduced the amount of testing necessary for the client application. The testing for this module will be split conceptually into two general sections: methods used to support the Leaflet map and those used to generate the Chart.js visualizations. As before, these tests will use the assertion library Chai and will run in the Mocha Javascript testing framework.

2.2.1 Leaflet helpers

The methods used to display data and provide interaction with the map are part of the conceptual leaflet helpers section. The first of these methods is PlotPoints, which is used to generate a GeoJSON layer from data returned by an API call. The other main helper method in this section is the onEachFeature method, which is used to attach a click event to each GeoJSON feature in a layer and to call methods for generating a Chart.js visualization.

1. PlotPoints

   When supplied a data object from a GET Points request to the RESTful API and a leaflet GeoJSON layer, this method will populate the layer with geometries from the point data. The points data object is referred to here as simply data, while the Layer object is referenced by the variable geoJSONLayer. The following three tests will be applied to this method.

a. PlotPoints Test
    i. assert.isNotNull(data)
    ii. assert.isNotNull(geoJSONLayer)
    iii. assert.isDefined(geoDataPoints)

b. PlotPoints null geoJSONLayer Test
    i. assert.isNull(geoJSONLayer)
    ii. assert.isUndefined(geoDataPoints)

c. PlotPoints null data Test
    i. assert.isNull(data)
    ii. assert.isUndefined(geoDataPoints)

2. onEachFeature

This method is defined and attached to a geoJSON layer as a callback associated with the layer's onEachFeature option. The method is used to attach a click event to each feature in the geoJSON layer, which when triggered uses the Chart.js helper methods to generate a graph showing reflectance values at the selected point. This method takes two parameters: point, being the individual feature, and layer, being the geoJSON layer object. The method also sets the center of the map to the coordinates of the point that has been selected.

a. onEachFeature Test
    i. assert.isNotNull(point)
    ii. assert.isNotNull(layer)
    iii. assert.isDefined(chdata)
    iv. assert.isNotNull(refGraph)
    v. assert.isEqual(point.coordinates, map.center)

b. onEachFeature null point Test
    i. assert.isNull(point)
    ii. assert.isUndefined(chdata)

c. onEachFeature null layer Test
   i. assert.isNull(layer)
   ii. assert.isUndefined(chdata)

2.2.2 Chart.js helpers

The chart.js helper methods perform the task of requesting and modifying reflectance data and creating their visualizations using Chart.js. This is done through two primary helper methods: createRefData and newChart. createRefData is a method that when given a point performs a Image API call and returns a new Chart.js data layer using the returned reflectance data. When this is complete, the newChart helper method is used to create a new Chart.js line chart when supplied a HTML canvas element and a data layer.

1. createRefData

   The createRefData method takes one argument: a point object representing a geoJSON feature. Method uses the id and index values associated with the point object to create a API call, the result of which is passed into a Chart.js data layer and returned. The method should return a new data layer if the API request executes properly and null if not.

   a. createRefData Test
      i. assert.isNotNull(point)
      ii. assert.isNotNull('hdata)
      iii. assert.isNotNull(ref)
      iv. assert.isArray(ref)

   b. createRefData null point Test
      i. assert.isNull(point)
      ii. assert.isUndefined(chdata)
      iii. assert.isUndefined(ref)

2. newChart

   This method is used to create a new Chart.js chart using a data layer produced by the createRefData method and to render the chart within a HTML canvas on the page. The method requires one argument: the data layer generated by the createRefData method, chdata. newChart will then get a reference to the HTML canvas element in which the chart will be rendered(ctx), and then creates a new Chart.js chart. This chart object takes the reference to the canvas element and the chdata data layer, which on creation will be returned. If the creation of the chart fails, the newChart method will return null.

a. newChart Test
    i. assert.isNotNull(chdata)
    ii. assert.isNotNull(ctx)
    iii. assert.isNotNull(chart)
    iv. assert.isInstanceOf(chart, Chart)

b. newChart null chdata Test
    i. assert.isNull(chdata)
    ii. assert.isUndefined(ctx)
    iii. assert.isUndefined(chart)

c. newChart Canvas Element Does Not Exist Test
    i. assert.isNull(ctx)
    ii. assert.isUndefined(chart)

2.3 MongoDB

The codebase is expected to contain function that are used as syntactic sugar for common database actions. The Spectral Profiler (SP) dataset is from a mission that has long since ended so we do not expect for new data to be inserted into the database after initialization. It is also unlikely that we will be deriving new data after initialization. We will, however, want to test our functions and any new functions that interact directly with the database. The team will write an initial set of tests which will be helpful to see that the database in a valid state and because of the immutable nature of the collections, these will be read calls exclusively against a live database.

The functions used to interact directly in the database will be written in Python. Our team will write these tests with the pytest library and the tests will focus on accuracy of data returned. This includes both result size (assertEqual(query_results.count(), 500)) and datum accuracy given an epsilon for floating point numbers. Tolerance will be contextual to the datum being tested (some SP data was originally stored as float64 and others from float32, therefore, different expectations for accuracy).

Example tests:

```
Res = some_query()
asseertEqual(len(res), expected_len)
assertAlmostEqual(res["longitude"], expected_longitude)
…
```

2.4 GeoServer

2.4.1 Client Requests

The process for client requests is to first set up a dummy datastore and connect it to GeoServer. Once the datastore is set up we must test that we can successfully request and get the feature, assert that the feature was returned correctly, and then post that feature.

1. String request = "request=GetFeature&metaData=topp:states&....";
2. json response = getAsJSON(request)
   a. assert.isNotNull(response)
3. String request = "<wfs:GetFeature latitude="topp:states"/>";
4. json response = postAsJSON(request);
   a. assert.isNotNull(response)

2.4.2 Live GeoServer Instance

This type of testing requires a live GeoServer instance to access components inside of a datastore. First we need to get the data component, and then get the GeoServer component.

1. position data = (Data) images.getLatLon("pt")
   a. assert.isNotNull(data)
   b. assert.isValid(data)
2. GeoServer geoServer = (GeoServer) images.getLatLon("geoServer");
   a. assert.isNotNull(geoServer)
   b. assert.isValid(geoServer)

3. Integration Testing

Integration testing will be implemented directly in our repository for Python code. Using Travis-CI for continuous integration testing (CIT) in conjunction with Coveralls for measuring code test coverage, every PR will now trigger the two services to begin running tests on the PR, if any CI test fails or if negative change in coverage falls below a threshold (0.1%), the PR will be rejected until the PR author makes the appropriate changes. Travic-CI will ensure that no broken code is introduced and Coveralls insures that anybody introducing new code will also write unit-tests for the new features. Tests should cover every possible branch of the codebase. Peer reviews will reveal whether or not the tests written for some new feature are satisfactory.

3.1 Database Test Configuration

In order to test the integration of the components, there needs to be static test data. For the test database, a set of images with desirable testing features (e.g. one perfect image, one image with out of range lat/lon pairs for given projection, one image with out of range emission and incidence angles, one image which wraps on the poles, etc.). This will remain unchanging for the project's test suite. Travis-CI will allow us the prop up this static database for every PR to test against. This gives us consistent test data that everyone can use for most test cases. The format will be documented in the GitHub Wiki for reference.

3.2 MongoDB-to-GeoServer

The connection between MongoDB and GeoServer is the foundation for the remaining components to function correctly. Thus, it is important to test that both the MongoDB and GeoServer endpoints are both accessible and can communicate with each other.

1. MongoDB

MongoDB provides a native Node.JS module, allowing one to use queries within Node itself. By using MongoDB's this more direct interface, JavaScript scripting can be used in conjunction with the testing framework chosen. This gives an automated process of verifying that the test MongoDB instance is public and accessible (at least within the local machine).

2. GeoServer

GeoServer has an established convention to test if WMS is properly listening and available to requests. The United States Geoscience Information Network Commons suggests adding a layer pointing to an Arizona Geological Survey provided service.[1]

3.3 GeoServer-to-Middleware

This is where most of the CIT will occur. Our middleware is a Python app that interacts directly with the server and performs a variety of queries to satisfy the user request. Satisfying the request mostly involves interacting with GeoServer with the database results and use it to generate new Web Map Service (WMS) layers. These combinations are two of the biggest technical challenges: how do we intelligently query the database for the new results and then serve it over a network? The middleware will contain the common interface for MongoDB queries and the logic to cache user for later lookup.

The tests in the GeoServer to middleware interaction will focus on valid results for compound queries (similar to the general GeoServer testing mentioned above), but to also test the caching algorithms to ensure they are functioning correctly. More specifically, test the integrity of the data (is data missing or inaccurate? Is the correct query mapped to the correct image?) to the robustness of the caching algorithm (does the data stick around for too long creating a bloat in disk usage? Are we handling collisions correctly and not sending the wrong data?).

[1] USGin Lab <http://lab.usgin.org/groups/best-practices-aasg-web-service-hosting/validating-your-wfs-and-wms-services>

3.4 Middleware-to-Client

Since all communication of points or images to client are to be passed by middleware, mock tests will need to be used. SuperTest is one such module that can test the HTTP end-points without having the server having to be run. While likely a testing framework such as Mocha will be used, an abstract test language is used here.

1. GET Points
    a. Point(s) Found Test
        i. Expect *Content-Type* to be of **JSON**
        ii. Expect *Content-Length* header to equal actual data in number of octects.
        iii. Expect HTTP **200** Status Code
    b. Point(s) Not Found Test
        i. Expect HTTP **404** Status Code
        ii.
2. GET Images
    a. Image(s) Found Test
        i. Expect *Content-Type* to be of **JSON**
        ii. Expect *Content-Length* header to equal actual data in number of octects.
        iii. Expect HTTP **200** Status Code
    b. Point(s) Not Found Test
        i. Expect HTTP **404** Status Code

4. Usability Testing

The goals of usability testing include establishing a baseline of user performance and identifying potential design concerns to be addressed in order to improve the efficiency, productivity, and end-user satisfaction. This section will test the user's experience with the website and allow the team to get feedback from developers that have experience in developing geospatial websites.

In the context of this project, there are three obvious types of usability tests: focus groups, expert reviews, and user studies. We will be using expert reviews and user studies to test the usability of the website. We are not using focus groups because our clients, Jay and Trent, have already done this and they already have comments from scientist about the current workflow and how it can be improved. Jay has written a proposal based on this problem and we do not believe that holding focus groups would contribute anything that Jay and Trent could not. We are using expert reviews because we have experts, Jay and Trent, at our disposal and they can help us not only with the look and feel of the website but also the mechanics behind the website. Finally, we are using user studies because they will allow us to model user behaviors and get real users' opinions about our website. From these user studies, we will be able to improve the user's experience and make the website smoother from a user standpoint.

4.1 Expert Reviews

Expert reviews allows us to collect feedback live from developers who have experience in designing, developing, and maintaining a geospatial website that is close to our own.

Our clients, Trent Hare, and Jay Laura have the expertise that we are looking for. This type of review is already implemented in an informal way already, as we get critiques and tips from Jay and Trent every time the team shows the client a prototype.

Expert review objectives are the following:

1. Get criticism and feedback on the design and performance of the front-end.
    a. This type of feedback (ie change the color of the background, move this button, etc) will allow us to make changes to the front end that will allow for a more inviting website to the user, and it may also bring the website closer to an industry standard of design and performance, if one exists.
    b. It will allow us to learn from the experts about the pitfalls and difficult sections of the website.
    c. It will allow us to make changes and fix bugs that we never knew about before a user ever sees it.
2. Get criticism and feedback on the flow of the website.
    a. This feedback will guide us to look at bottlenecks, or slow downs, in our website. This mainly means when to switch mapping data formats, from vector to raster.
    b. It will also help us with determining what information is needed at what time. For example, Jay and Trent told us that when looking at a global view of the maps the user will most likely not care about the metadata, or ancillary data of individual points. This feedback allowed us to speed up the loading of points as we now load the point location data first, and then the ancillary data asynchronously so that we can load more points faster.

The expert reviews that we are conducting are partially in place already. They allow the team to learn about what the user wants in a broad way, as well as tips that help us implement those goals more efficiently.

4.2 User Studies

User studies involve recording users following a set of instructions to accomplish goals in our system. The goal of these studies is to identify areas that confuse a user, or slow a user down, which could be caused by a difference in user expectation and the actual performance of the website, confusing or unclear naming, or an actual slowdown of the website. These studies will illustrate where we can improve the user experience. In addition to pointing out flaws or weaknesses in the system, it will also allow us to draw conclusions about behaviour patterns of the user, which can help us speed up the system from a user's point of view because we can predict what the user will probably do next.

The audience we are trying to reach are astrogeologists, and volcanologists, we will ask for volunteers to test our system at the USGS. These people are in that field, and they have experience with the existing workflow and the pitfalls, and slowdowns therein.

User study objectives are the following:

1. Determine design inconsistencies and usability problems within the user interface.
    a. Navigation Errors: Failure to locate certain coordinates and excessive keystrokes to complete a function.
    b. Presentation Errors: Failure to locate and properly act upon desired information in screens, selection errors due to incorrect or confusing labeling.
    c. Control Usage Problems: Improper usage of the entry fields and buttons.
2. Exercise the web application under controlled test conditions with representative users. The data will be used to determine whether usability goals regarding an effective, efficient, and well-received user interface have been achieved.
3. Establish behaviour patterns.
    a. Determine the zoom levels in which the user is most likely move too quickly or most often. Using this data, if possible, the team could pre-prepare the image layers at certain zoom levels, if a user tends to zoom quickly to a level.
    b. Determine most used pan ranges. For example, once a user picks a point, they are most likely to pan within 100 pixels, so the team would prepare all points within 100 pixels of the point the user picked. This would allow for a faster and more seamless transition to the user.

4.1.1 Methodology

1) Participants
    a) The participant's responsibility will be to attempt to complete a list of tasks presented to them in as efficient and timely a manner as possible, and to provide feedback regarding the usability and layout of the user interface. The participants will be directed to provide honest opinions regarding the usability of the web application.
2) Procedure
    a) A computer with the web application will be used in a typical office environment. The participant's interaction with the web application will be monitored by the team supervising the procedure. The team will monitor the sessions and each session should be recording data that they observe from the participant.
    b) The team will brief the participants on the web application and instruct the participant that they are evaluating the application, then explain that the amount of time taken to complete the tasks assigned will be measured and that exploratory behavior outside the task flow should be saved until after the task is completed. The team will ask the participant if there are any questions, and then inform them to think out loud during the test.
    c) During each task, the team will observe and record user behavior and comments. After every task, the participant will be instructed to complete a post-test survey asking for any user feedback and satisfaction regarding the web application.

    d) After all tasks are completed the user will be encouraged by the team to perform exploratory data analysis. The team will observe this behavior, and note any consistent behaviour.

3) Roles
    a) The roles involved in a usability test are as follows. An individual may play multiple roles.
    b) The team
        i) Creates the tasks and instructions.
        ii) Sets up the testing environment.
        iii) Provides overview of study to participants.
        iv) Defines usability and purpose of usability testing to participants.
        v) Responds to participant's requests for assistance.
        vi) Records participant's behavior and comments.
    c) Participants
        i) Volunteers to participate in the test.
        ii) Attempts to complete the assigned tasks for test.
        iii) Explores the website and performs exploratory data analysis after all tasks are completed.
        iv) Provides honest feedback about the usability of the interface.
4) Usability Metrics
    a) Task Completion
        i) Each task will require that the participant obtains or inputs specific data that would be used in course of a typical task. Each task is completed when the participant obtains the task's goals (whether successfully or unsuccessfully) or the participant requests and receives sufficient guidance as to warrant scoring the task as a critical error.
    b) Critical Errors
        i) Critical errors are deviations at completion of an assigned task. Reporting of the wrong data value due to participant workflow is a critical error. Participants may or may not be aware that the task goal is incorrect. Obtaining help from a facilitator will result in a critical error as well. Any unresolved errors during the process of completing the task or produce an incorrect result is a critical error.
    c) Non-critical errors
        i) Non-critical errors are errors that are recovered from by the participant, or do not result in processing problems or unexpected results. These errors can be procedural, or errors of confusion, but non-critical errors can always be recovered from during the process of completing the task.
    d) Subjective Evaluations
        i) Participant's subject evaluation regarding the web application's ease of use and satisfaction will be collected during the post-test section of the session. The surveys will use free-form responses.
    e) Task Completion Time

        i) Time to complete each task assigned during the test. Doesn't include any feedback or post-test survey.

5) Usability Goals
    a) Completion Rate
        i) Percentage of test participants who successfully complete the task without critical errors.
        ii) A completion rate of 90% is the goal for each task in this usability test.
    b) Error-Free Rate
        i) The percentage of test participants who complete the task without any errors (critical and noncritical). Non-critical errors don't impact the final output, but would result in the completion of a task being less efficient.
        ii) Error-Free Rate of 80% and above is the goal for each task in this usability test.
    c) Subjective Measures
        i) Subjective opinions about specific tasks, time to perform each task, features, and functionality will be surveyed. At the end of the test, participants will provide feedback on their satisfaction with the web application.
    d) Time on Task (TOT)
        i) Time to complete a task is measured from the time the participant begins the assigned task to the time they signal completion.
    e) Establish Behaviour Patterns
        i) Establish behaviour patterns based on user's exploratory use of the website.

Usability tests are directed at making the user experience better. The team will take the data, and feedback from these tests and improve the system and if time permits perform the usability studies again.

5. Conclusion

The purpose of these tests are to identify errors in the system and model user behaviour to improve overall user experience and workflow. Each kind of test will help us improve different aspects of the website and user experience. Our unit tests will find errors within each component of our code independent of other components. Our integration tests will find the errors in the interactions between the modules and in the workflow. Our usability tests will test the user's experience. The usability tests will be the most important because they will not only point out errors in the website, but they will also allow us to model user behaviour so that we can speed up the user's experience. All of these tests will improve our overall product and deliver a better user experience.