Solar Plant Fault Detection and Diagnostics Software:

Following Guidelines of the NASA Systems Engineering Handbook

Scott LaForge (Team Leader) Eric Carestia
Andon Coleman (Chief Architect) Justin Hodnett
Chris McCoy Jorge Ocañas
Chris Steiner Josh Yanai
Janusz Zalewski, PhD (Team Mentor)
Abstract

The objective of this project is to apply guidelines of the NASA Systems Engineering Handbook to the development of software for a selected problem in NASA Ground Operations technical area. The problem selected for this project is fault detection and diagnostics for a solar plant. This document described the development of a software package called FINEMAN (Fault Identification for Nasa Exploration Missions And Navigation), including software requirements specification, software design, implementation and testing. The FINEMAN software logs on automatically to the testing platform, which is the Florida Gulf Coast University solar power plant, acquires and records data in real time, and conducts reasoning about potential faults using a technique called Bayesian belief networks.
# Table of Contents

1. Introduction 3

2. Project Overview 4
   2.1 Testing Platform – FGCU Solar Power Plant 5
   2.2 Fault Model 7
   2.3 Choice of the Fault Identification Technique 8
   2.4 Applicability of NASA Systems Engineering Handbook to Software 9
   2.5 Educational Considerations 10

3. Software Requirements Specification 11
   3.1 Definitions and Acronyms 11
   3.2 Initial Assumptions 11
   3.3 Application of the Handbook 12
   3.4 List of Requirements on the Client 13
   3.5 List of Requirements on the Extraction Tool 15
   3.6 List of Requirements on the Injection Tool 17

4. Software Design Description 21
   4.1 Functions Summary 21
   4.2 Application of the Handbook 21
   4.3 Design of Fineman Client 22
   4.4 Design of Fineman Tools 24

5. Software Implementation 27
   5.1 Application of the Handbook 27
   5.2 Connectivity Component of the Fineman Client 27
   5.3 Reasoning Component of the Fineman Client 27

6. Software Testing 34
   6.1 Application of the Handbook 34
   6.2 Testing the Connectivity Component 34
   6.3 Testing the Reasoning Component 35
   6.4 Test Plan for the Extraction Tool 38
   6.5 Testing the Injection Tool 39

7. Conclusion 42

8. References 43

Appendices 46
1 Introduction

The objective of this project is to apply guidelines as outlined in the NASA Systems Engineering Handbook [1] to the development of software for a selected problem in NASA Ground Operations technical area. This is an innovative approach, since to our knowledge there has not been any material published, yet, on using this specific Handbook for software development.

The problem selected for this project is fault detection and diagnostics for a solar plant. Solar energy is widely used on NASA missions, for example, on Space Station, and automatic fault detection and diagnostics is of great importance to determine early causes of failures and indentify parts, which failed, so the specific mission could continue. As a general problem of vehicle health management, such software can also help in prognostics of potential failures. The added value of developing such software is that it can be also used in regular solar power plants, not necessarily those related to NASA missions.

The rest of this report is organized as follows. First, in Section 2, a general overview of the project as well as its limitations are given. The next sections present, in this sequence, the requirements definition (Section 3), design description (Section 4), software implementation (Section 5), and testing (Section 6). This is followed by a conclusion (Section 7) and references (Section 8).
2 Project Overview

In NASA missions, both manned and unmanned, it is essential to provide power to the space vehicle, and solar panels are the major technology that enables this. To operate solar panels, continuous monitoring of performance and health of the installation is needed to ensure uninterrupted operation and safety of the mission, both before and during launch, as well as in flight and during landing. It is essential that Ground Operations has all the necessary information to identify and predict any actual and potential faults to prevent or mitigate prospective failures.

To ensure timely identification of faults in solar plant equipment, we develop a software package, named Fineman, which tracks all necessary plant parameters in real time, by downloading their measured values directly from the plant, and based on these measurements reasons about potential faults and their causes, by using an artificial intelligence technique called Bayesian belief networks.

The following sections outline the rationale for the major decisions made in this project, that is:

- The choice of a solar power plant used to test the Fineman software (Section 2.1).
- The choice of the fault model to use in computations (Section 2.2).
- The choice of the artificial intelligence technique to reason about faults (Section 2.3).
- The applicability of the NASA Systems Engineering Handbook (Section 2.4).

Since all of the student team members are new to this sort of application, as well as to fault diagnostics techniques, artificial intelligence methods, and the Handbook, respective methods and processes of educating the team in related technologies are presented in Section 2.5.
2.1 Testing Platform – FGCU Solar Power Plant

In every system or software engineering project in the academic environment, it is essential to make the project a real-life experience and give the students a sense of a practical application. Since access to the actual data set from a live or past NASA mission would be difficult to obtain within the time period required, a decision was made to utilize data from a recently built solar power plant at Florida Gulf Coast University [2-4]. The plant is operated at FGCU and the performance data are collected by a Sentalis data logger installed by and maintained by Draker Labs [3], as illustrated in Figure 1.

The diagram also illustrates how the *Fineman* software developed in this project will be used. Upon a request from a user (operator), it will login upon request to the Draker Labs website, given appropriate login permissions (username and password), and retrieve measurement data collected by the Sentalis data logger, downloading them into a .CSV file. The measurement data recorded in the file can then be analyzed by a specialized part of *Fineman*, to determine whether any faults have occurred and/or what are their causes.

Figure 1: Physical Diagram of the System.
The devices whose measurement data are collected by the Sentalis data logger are presented in Figure 2. They include pyranometers, power meters, inverters, transformers, weather station, and other relevant information. The Fineman software has only access to the Sentalis data logger and does not interfere with the data collection by the data logger.

Figure 2: Sentalis High Performance Data Acquisition and Monitoring System [3].
(the blanked out subsystem is offline seasonally).

The format of a sample .CVS file, which is supplied by the data logger is presented in Appendix 1. It illustrates the solar plant parameters whose values are measured and recorded for one building. There are three buildings total, for which power is generated and supplied, with multiple solar pads for each building.
2.2 Fault Model

There are two issues to be considered at the beginning, regarding the identification of potential faults. First, since there are multiple sources of potential failures, all of them have to be listed beforehand and presented to the developers as an assumption. It is also assumed that ranges of correct values are known for all continuous measurement variables and all correct states are known for discrete variables.

Since the Fineman software will be retrieving measurement data and deriving decisions on potential failures in real time, it has to be prepared for all sorts of circumstances that may arise during operation of the plant. However, during normal operation failures may occur rarely or not occur at all, therefore it would be difficult to test the software for all operating conditions. In this view, an offline tool is needed, which would assist in testing the Fineman software by simulating potential faults. The architecture of such a “fault injector” is presented in Figure 3.
2.3 Choice of the Fault Identification Technique

Once the solar plant measurement data are stored in the .CSV file, a respective technique can be used to analyze the data and conduct inference regarding the plant’s status and reliability of its respective components. There are several techniques which can be used to analyze measurement data and determine faults, among them those based on continuous models of the plant, such as Kalman filters or particle filters, and those based on the discrete models of the plant, such as rule-based techniques, state machines, network techniques, and queuing theory techniques. Taking into consideration three factors:

- the diversity of measured parameters, some of which are presented in Table 1
- the possible simplicity of the technique, so it can be mastered by a senior within weeks,
- the availability of inexpensive (possibly, public domain) tools for the method

it has been decided to use Bayesian belief networks as a reasoning technique [5], and its corresponding tool called Netica [6].

Table 1. Sample Devices Monitored for Faults.

<table>
<thead>
<tr>
<th>Specific device</th>
<th>Example failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>LI-200 SA Pyranometer Sensor</td>
<td>Reading out of range</td>
</tr>
<tr>
<td>Shark 100 Power Meter</td>
<td>Reading unavailable</td>
</tr>
<tr>
<td>PowerGate Plus 500 kW Inverter</td>
<td>Thermal failure</td>
</tr>
<tr>
<td>PowerGate Plus 250 kW Inverter</td>
<td>Thermal failure</td>
</tr>
<tr>
<td>WXT 520 Weather Transmitter</td>
<td>Uncorrelated wind and ambient air temperature</td>
</tr>
<tr>
<td>CR 1000 Datalogger</td>
<td>Packet losses</td>
</tr>
</tbody>
</table>

A Bayesian belief network (BBN) relies on applying a Bayesian inference to a graphical representation of a problem in a form of a hierarchical network [5], with nodes representing random variables and directed arcs representing probabilities (beliefs) about the dependencies (relationships) among these variables. Each node has a number of states, with some probability distribution over the states. Each arc represents a conditional probability reflecting dependencies on the predecessor nodes. Reasoning in a BBN relies on making assumptions on values of respective variables (which can be measured) and analyzing their dependencies to derive conclusions about the state of other variables in the network, which may represent faults.
2.4 Applicability of NASA Systems Engineering Handbook to Software

Given the environment and the methods for software development as described above, appropriate software development standards need to be followed to ensure appropriate quality of the software product. Traditionally, standards developed by professional societies, such as IEEE [7-8], are usually followed in software development. However, for this project a NASA Systems Engineering Handbook has been selected as a development guideline, with the assumption that more specific recommendations may follow from IEEE standards.

NASA Systems Engineering Handbook is a very comprehensive document comprising some 340 pages of information and guidelines on development and implementation of space related systems. In a 15-week semester, several assumptions had to be made to target a realistic software development and still meet the criteria of realistic software to be developed.

In this view, it was assumed that a simple waterfall model of software development will be followed, composed of the following four stages: Software Requirements Specification, Software Design, Software Implementation, and Software Testing, accompanied by respective techniques of Quality Control. Consequently, a decision has been made to apply the following sections of the Handbook:

- Section 4.2 Technical Requirements Definition, with respect to Software Requirements Specification
- Section 4.4 Design Solution Definition, with respect to Software Design
- Section 5.1 Product Implementation, with respect to Software Implementation, and
- Section 5.3 Product Verification, with respect to Software Testing.

Quality Assurance processes, in addition to Software Testing, are implemented using specific techniques at each stage, and involve stage specific activities, such as Requirements Validation, Design Reviews, and Code Inspection.
2.5 Educational Considerations

In a senior design project, where students’ prerequisite knowledge is essential for successful project completion, a number of factors that affect the final product need to be considered. In this particular project the student team had to be educated on multiple issues related to proper knowledge acquisition that could be useful in software development.

First, it was necessary to acquire the necessary domain knowledge on solar power plants none of the team members had before, in particular, related to data acquisition. For this purpose multiple references were identified regarding the use of data acquisition systems in solar plants [9-19], and some of them studied with the goal in mind to understand the measurement processes.

Second, respective literature was identified and studied in relation to fault management and health monitoring in solar power plants over the last decade [20-31]. In addition, respective lectures on Health Management and System Prognostics were prepared and given, following Team Mentor’s fellowship at the NASA Ames Center [32].

Third, the solution methodology selected, Bayesian belief networks, was discussed extensively in class, in a special module called “Prognostics with Bayesian Belief Networks” [32]. The tool to implement this technique, Netica [6], was selected from an array of other Bayesian network tools, and was presented and discussed in class.

Finally, given the theoretical background was acquired, as discussed in previous paragraphs, software engineering processes necessary to develop and implement the software had to be learned. Professional standards for software development were previously known to the development team from a Software Engineering Fundamentals class, which is a prerequisite to the Senior Software Project course. However, new information related to the NASA Systems Engineering Handbook had to be covered in a separate class module [32].

This background was necessary and sufficient to develop the Fineman software as described in the following sections.
3 Software Requirements Specification

3.1 Definitions and Acronyms

CSV. Comma Separated Value. A file format for recording data.

BBN. Bayesian Belief Network. A method of reasoning about plant behavior.

Draker Labs website. Website of the following URL, specific to Florida Gulf Coast University solar power plant: http://solarems.net/

FINEMAN (Fineman). Fault Identification for Nasa Exploration (manned) Missions And (unmanned) Navigation. Name of the software package developed in this project.

GUI. Graphical User Interface.

Solar Plant. For this project, it is a solar power plant located at Florida Gulf Coast University. In general, it can be any solar power plant, but a respective Bayesian belief network will have to rebuilt for it.

3.2 Initial Assumptions

Requirements listed in this section have been developed for the physical representation of the system as shown in Figure 1, with the assumption that the Fineman software interacts only with the following entities, as illustrated in Figure 1:

- The plant's data logger, via a communication network (Internet), as shown in Fig. 2.
- The local database to store .CSV files.
- The user (operator), who has access to the Fineman software from the local computer.

Additional design and implementation assumptions are as follows:

- The Fineman software will run on the local PC under Windows XP/Vista/7.
- The Fineman software will consist of two parts: (1) the client, which will actually retrieve measurement data from the data logger and conduct respective reasoning, and (2) the tool, which will extract respective measurement data from the .CSV file and inject faults in it for the purpose of testing the client.
3.3 Application of the Handbook

The Software Requirements Specification stage is a process in software development, which receives input from the Customer (who is the only Stakeholder in this project), and produces output to the Software Design stage.

Of the seven items of Requirements Metadata (p. 48) in the Handbook, we only use three across this specification: ID, Owner (which we call here Source), and Verification Method, plus an additional item marked Notes, depending on particular needs. Since the Verification Method is common for all requirements, it is explained first, in the following paragraph.

Every requirement is formally verified according to the following four criteria:

- **Clarity**, understood as the property of the specification document that ensures its understanding and non-ambiguous interpretation by the reader proficient in the specification language.
- **Correctness**, the property of the specification document that ensures this document's compliance with external requirements, such as related documents, standards, scientific knowledge, common sense, etc.
- **Consistency**, the property of the specification document which ensures that there is no contradictory information in this document.
- **Completeness**, understood as the property of the specification document that ensures inclusion of all the information necessary to develop a specified system.

Initial formulation of requirement is followed by the Requirements Validation activity, which is formal process of applying these four criteria. As a result, the requirements Validation Document is produced, which point out all the defects, after which the requirements are revised and refined.
3.4 List of Requirements on the Client

3.4.1 Functional Requirements

Requirement 1
The *Fineman* software shall establish a connection to the Draker Labs website.
Source: Subteam #1.

Requirement 2
Once successfully connected, the *Fineman* software shall stay connected to Draker Labs website.
Source: Subteam #1.

Requirement 3
The *Fineman* software shall download a current CSV file from the Draker Labs website.
Source: Subteam #1.

Requirement 4
The *Fineman* software shall download the CSV file using the current date and time as parameters.
Source: Subteam #1.

Requirement 5
The *Fineman* software shall maintain a persistent configuration for every instrument in the system, including manufacturer information, measured readings and their valid values, and dependencies on other instruments.
Source: Subteam #4.

Requirement 6
The *Fineman* software shall summarize the health of the system in a concise graphical format, allowing an end-user the option of exploring the measurements of each instrument, faulty readings and their probable causes and manufacturer information.
Requirement 7
The *Fineman* software shall use a Bayesian Belief Network technique with the intention of inferring the status of instruments given an incomplete set of data.
Source: Subteam #4.

3.4.2 Reliability Requirements

Requirement 8
The *Fineman* software must recognize instruments that are purposely offline, to accommodate the fact that parts of the solar plant are offline seasonally.
Source: Subteam #4.

Requirement 9
The *Fineman* software shall operate continuously downloading a new CSV file every hour, or some other predetermined time.
Source: Subteam #1.

3.4.3 Interface Requirements

Requirement 10
The *Fineman* software shall have a GUI as presented in Fig. 4.

*Note 1.* The text boxes named Email and Password allow the user to enter their email and password for the Draker Labs website.

*Note 2.* The button named Change Directory serves the purpose of changing the directory to save the CSV file in.

*Note 3.* The button named Submit, when pressed, serves the purpose of sending the user’s login information to the Draker Labs website.
Source: Subteam #1.
Requirement 11
In response to pressing the submit button, the software shall download the .CSV files continuously as per Requirement 5.
Source: Subteam #1.

Requirement 12
The Fineman software shall terminate and exit upon pressing the X close window button in the right upper corner of the GUI in Figure 1, saving all data currently being downloaded, if the button is pressed in the middle of the downloading operation.
Source: Subteam #1.

3.5 List of Requirements on the Extraction Tool

The purpose of this tool is to extract sensor data from a comma-separated value file (.CSV). The software will then take the data and allow an end-user to select a specific instrument / data from a specific sensor to view and store it in a separate file.
3.5.1 Functional Requirements

Requirement 13
The Extraction Tool shall allow a user to extract selected sensor information from the .CSV file and save it into a different .CSV file.
Source: Subteam #2.

3.5.2 Interface Requirements

Requirement 14
The Extraction Tool shall have a GUI as presented in Figure 5.
*Note.* Descriptions of functions of respective buttons are included in the Figure.
Source: Subteam #2.

Figure 5: Sensor Selection User Interface with Explanations.
3.5.3 Data Requirements

Requirement 15
The data shall be collected from the .CSV file and displayed in the GUI data field as per Figure 5.
Source: Subteam #2.

3.6 List of Requirements on the Injection Tool

The purpose of this tool is to design a software program that shall simulate faults in a selected monitoring device of the Solar Energy Monitoring System by changing values in the device output file. A user interface should be able to select the monitoring device and faults to inject.

3.6.1 Functional Requirements

Requirement 16.
He fault Injection Tool shall alter measurement data in the requested .CSV file.
Source: Subteam #3.

Requirement 17.
The fault Injection Tool shall make an identical copy of the .CSV file into which injections may be inserted once the .CSV file is retrieved.
Source: Subteam #3.

Requirement 18.
The fault Injection Tool shall have access to a file containing all the standard ranges possible for each sensor and its components.
3.6.2 Interface Requirements

Requirement 19
The fault Injection Tool shall have a GUI interface allowing users to retrieve data from a .CSV file, as illustrated in Figure 6.
Source: Subteam #3.

![Figure 6: GUI for the Fault Injection Tool.](image)

Requirement 20. Sensor choice selection menu
The fault Injection Tool interface shall have a selection box, allowing the user to choose which sensor will receive a fault injection.
Source: Subteam #3.
Requirement 21. Standard sensor range display
In the fault Injection Tool display of the sensor’s standard ranges shall be displayed on the left-hand side of the screen based on which sensor was selected and the number of sensor components.
Source: Subteam #3.

Requirement 22. Numerical fault value input box
In the fault Injection Tool parallel to each component’s range on the right, a text box shall be displayed, allowing the user to insert a numerical fault value.
Source: Subteam #3.

Requirement 23. Fault value variance input box
In the fault Injection Tool to the right of numerical fault text box shall be a text box allowing the user to input the amount of variation from the fault chosen (ie. -5 to +5 from fault value).
Source: Subteam #3.

Requirement 24. Frequency of injection selection menu
In the fault Injection Tool, to the right of variance input text box shall be a selection box allowing the user to choose the frequency for which the fault shall be injected.
Source: Subteam #3.

Requirement 25. Frequency of injection selection menu choices
In fault Injection Tool the selection box for frequency shall include:
a. one time only
b. every 15 minutes
c. once per hour
d. once every 4 hours
e. once per day
Source: Subteam #3.
Requirement 26. Save Injection Parameters Button
The fault Injection Tool shall contain a button allowing the user to save all injection parameters in the upper left corner, once faults are chosen for any number of sensors.
Source: Subteam #3.

Requirement 27. Save Injection Parameters Button
The fault Injection Tool shall have a pop up window allowing the user to choose a name and a location for the fault injection parameters.
Source: Subteam #3.

Requirement 28. Load Injection Parameters Button
The fault Injection Tool shall have a load parameters button in the upper left corner to the right of the save parameters button that shall allow the user to populate the injection program with a saved injections parameter file.
Source: Subteam #3.

Requirement 29. Exit Button
The fault Injection Tool shall contain a button allowing the user to exit the program in the upper right corner.
Source: Subteam #3.

Requirement 30. Start Injection Button
In fault Injection Tool at the bottom of the screen shall be a start button allowing the user to begin injecting faults.
Source: Subteam #3.

Requirement 31. Terminate Injection Button
In fault Injection Tool at the bottom of the screen, to the right of the Start button shall be a terminate button allowing the user to terminate the fault injections.
Source: Subteam #3.
4 Software Design Description

4.1 Functions Summary

The *Fineman* software fetches instrument readings from a central Sentalis data logger, and presents it for the analysis. Next the data are analyzed using a statistical modeling to determine what sensors or circumstances may be to blame for all detected faulty readings. Then the software presents these findings to a user in a human-readable format.

In order to model the cause and effect relation between instrument readings, a high-level description of instrument connectivity is essential. For this project, it is understood that no cyclic relationships will exist in the description of instrument connectivity. This allows the application of a common technique in fault detection, known as a Bayesian Belief Network. A Bayesian Belief Network can be used to predict the probabilities of events or states given a partial view of a system, and this information can then be used to intelligently determine cause and effect relationships.

4.2 Application of the Handbook

The Software Design stage is a process in software development, which receives input from the Software Requirements stage, and produces output to the Software Implementation stage.

The designs are formally verified during the Design Review according to the following three criteria:

- *Cohesion*, defined as a measure of the strength of association of the elements within a module [33].
- *Coupling*, defined as a measure of the interdependence among modules in a computer program [33].
- *Traceability*, defined as the degree to which each element in a software development product establishes its reason for existing [33].
4.3 Design of Fineman Client

There are two essential components of the client’s design: the Connectivity Component, whose function is to maintain connection with the Sentalis data logger, and the Reasoning Component, whose function is to determine, using a Bayesian belief network model, whether there are any potential failures instrumentation in the solar power plant and what are their causes.

The Connectivity Component can be modeled as a concurrent module running an infinite loop until requested to exit.

![Flowchart of the Connectivity Component](image)

Figure 7: Basic Flowchart of the Connectivity Component.

The detailed design of the Connectivity Component is rather involved and presented in Appendix 2. It is adapted from an example on logging into a website by HTTP client from Apache. This design skeleton shows the essential steps in logging on using the HTTP protocol. It has to be significantly refined for the implementation, because the Draker Labs website use HTTPS protocol (secure HTTP).
The Reasoning Component makes use of the solar plant measurement data modeled as a directed acyclic graph (DAG). A useful property that makes the application of a Bayesian Belief Network practical for fault analysis in this system is that the location of an instrument on the DAG determines its set of potential causes (causality) and effects (propagation), in terms of connected instruments, as shown in Figure 2. Given a subset of measured data, the Bayesian Belief Network predicts the probability each instrument will exist in one of many discrete states. These predictions are analyzed against actual conditions to infer a faulty measurement’s causality, as well as the set of instruments whose operation may be negatively impacted by this measurement.

To visualize the process by which the Reasoning Component analyzes the health of the solar power system, it is helpful to imagine the pipeline presented in Figure 8.

![Diagram of Conceptual Pipeline Stages of the Reasoning Component](image)

Figure 8: Conceptual Pipeline Stages of the Reasoning Component.

Each step in this pipeline is described below.

**Step 1.** The software parses a configuration file that stores information on each of the instruments in the Sentais data logged environment. This information includes the manufacturer, model number, serial number, and connectivity for each of the instruments, as well as a list of measurements and a range of valid values the instrument will generate. Additionally, many instruments have operational constraints that may be validated by examining readings from other instruments; these are dependent measurements, and the configuration also keeps track of these for every instance of an instrument.

**Step 2.** Given a subset of measured values, the software identifies the instrument(s) that generated or depend on each, and performs sensor validation. If all instruments are determined to be within operational limits, the software skips the following pipeline stages and proceeds to the final stage (user presentation).
Step 3. Before using the input data to make inferences on the health of the system, the overwhelming number of measurements must be stripped down to the essentials. The preprocessor clusters related measurements together and summarizes their status in order to reduce the complexity of fault modeling.

Step 4. After the input data is simplified, it is plugged-in to a Bayesian Belief Network that models the relationship between instrument measurements and instrument health. The software uses the BBN to infer the status of instruments given a subset of known measurements.

Step 5. Once the BBN has calculated the probabilistic state of each instrument in the system, the software uses the information to predict the causes of any detected faults, and optionally, the affect a faulty reading may have on other instruments.

Step 6. Finally, the software combines all of the information into a human-readable report that includes the health of each instrument, the values for every input measurement, and for each faulty measurement, a list of likely causes.

4.4 Design of Fineman Tools

For both tools, the Extraction Tool and the Injection Tool, detailed design diagrams are presented in Figures 9 and 10, respectively.

The class diagram in Figure 9 shows the structure of the Extraction Tool. It is composed of a MainWindow class, which uses Resources and CsvParser classes to fulfill respective requirements.

The Injection Tool is represented by a flowchart in Figure 10. Its behavior is straightforward due to the sequential nature of the actions specified in the requirements.
Figure 9: Extraction Tool Class Diagram.
Figure 10: Flow Chart of the Extraction Tool.
5 Software Implementation

5.1 Application of the Handbook

The Software Implementation stage is a process in software development, which receives input from the Software Design stage, and produces output to the Software Testing stage. The implementation is formally verified during the Test Cases described in Section 6. The current section describes on the code.

The implementation code for both the Fineman client and both tools is pretty extensive and is not discussed in this report in its entirety due the lack of space. We present only selected part of the client’s implementation.

5.2 Connectivity Component of the Fineman Client

The client’s Connectivity Component requires the following library needs to be added to the classpath: htmlunit [34-35]. The user has to have a keystore that has the certificates for the websites security connection. Instructions on how to create and where to place that file are available from the Oracle website [36]. After that all there is left is to compile and run the component.

When the Fineman software runs, the operator enters their email and password for the Draker Labs website, chooses a directory to save the file to and clicks submit. The user then can minimize the window and the program will continue to run, downloading a .CSV every hour.

5.3 Reasoning Component of the Fineman Client

For the Reasoning Component, sample implementation fragments are separated into three sections below, according to stages outlined in Figure 8: first, Configuration Parser, Input Validator and Pre-Processor stages, then Belief Network and Post-Processor stage, and finally the User Presentation stage.
5.3.1 Configuration Parser, Input Validator and Pre-Processor Stages

Samples of the general interface used by the configuration parser, input validator and statistical pre-processor, for the C++ API in Netica software are presented in Figures 11 through 14.

```cpp
sfd_Inverter* inverter_A = new sfd_Inverter500 ("Pad A Inverter");
sfd_Inverter* inverter_B1 = new sfd_Inverter250 ("Pad B1 Inverter");
sfd_Inverter* inverter_B2 = new sfd_Inverter250 ("Pad B2 Inverter");
sfd_GenerationMeter* gen_A =
    new sfd_GenerationMeter ("Pad A Shark Meter", inverter_A);
sfd_GenerationMeter* gen_B1 =
    new sfd_GenerationMeter ("Pad B1 Shark Meter", inverter_B1);
sfd_GenerationMeter* gen_B2 =
    new sfd_GenerationMeter ("Pad B2 Shark Meter", inverter_B2);
sfd_LoadMeter* load_AB7 = new sfd_LoadMeter ("AB7 Load Meter");
inverter_A->add_dependency <float> ("Temperature", ambient_temp, -20.0f, 50.0f);
inverter_A->add_reading <float> ("AC Frequency", Hz, 59.5f, 60.5f);
inverter_A->add_reading <float> ("AC Power", "KW", 0.0f, 500.0f);
inverter_A->add_reading <float> ("AC Voltage", "VAC", 183.0f, 229.0f);
inverter_B1->add_dependency <float> ("Temperature", ambient_temp, -20.0f, 50.0f);
inverter_B1->add_reading <float> ("AC Frequency", Hz, 59.5f, 60.5f);
inverter_B1->add_reading <float> ("AC Power", "KW", 0.0f, 250.0f);
inverter_B1->add_reading <float> ("AC Voltage", "VAC", 183.0f, 229.0f);
inverter_B2->add_dependency <float> ("Temperature", ambient_temp, -20.0f, 50.0f);
inverter_B2->add_reading <float> ("AC Frequency", Hz, 59.5f, 60.5f);
inverter_B2->add_reading <float> ("AC Power", "KW", 0.0f, 250.0f);
inverter_B2->add_reading <float> ("AC Voltage", "VAC", 183.0f, 229.0f);
gen_A->add_dependency <float> ("Temperature", ambient_temp, -20.0f, 70.0f);
gen_A->add_dependency <float> ("AC Power", inverter_A, 0.003f, 205.0f);
gen_A->add_dependency <float> ("AC Voltage", inverter_A, 90.0f, 265.0f);
gen_B1->add_dependency <float> ("Temperature", ambient_temp, -20.0f, 70.0f);
gen_B1->add_dependency <float> ("AC Power", inverter_B1, 0.003f, 205.0f);
gen_B1->add_dependency <float> ("AC Voltage", inverter_B1, 90.0f, 265.0f);
gen_B2->add_dependency <float> ("Temperature", ambient_temp, -20.0f, 70.0f);
gen_B2->add_dependency <float> ("AC Power", inverter_B2, 0.003f, 205.0f);
gen_B2->add_dependency <float> ("AC Voltage", inverter_B2, 90.0f, 265.0f);

/**< The entire B2 subsystem is offline parts of the year... */
inverter_B2->set_flag (INSTRUMENT_CASCADED_OFFLINE);
gen_B2->set_flag (INSTRUMENT_OPTIONAL);
```

Figure 11: Sample Implementation of the Configuration Parser (setup.cpp)
Figure 12: Sample Implementation of the Input Validator (instrument.h)
Figure 13: Sample Implementation of the Input Validator (instrument.cpp).
Figure 14: Illustration of a Pipeline Data Between Input Validation and Pre-processing.
5.3.2 Belief Network and Post-Processor Stages

The Bayesian Belief Network in Figure 15 illustrates the far-reaching implications ambient temperature has on the operation of the system, and the ability to intelligently predict measurements given only a subset of related data.

![Figure 15. Sample Bayesian Belief Network Using the Netica API.](image)

Nodes in grey represent nodes whose measured values were supplied, while brightly shaded nodes represent the predicted probabilities given the set of supplied measurements.

This network also illustrates limitations imposed by software licensing, where at most 15 nodes may be modeled in any given case set. The actual web of dependencies is much more connected than can be modeled in the current implementation.

After all available data are modeled using one or more Bayesian Belief Networks, it undergoes statistical post-processing to prioritize fault causality, and is summarized in an intuitive graphical format as illustrated in the user presentation stage as illustrated in Figure 16.
5.3.3 User Presentation Stage

To illustrate the inner working of the Fineman Reasoning Component, as presented in Figure 16, recall from Figure 2 that generation meters A, B2, and B1 are connected in serial on a single RS-485 line, and that B2 is situated to the right of B1 on the diagram. The cable connecting B2 and B1 appears as a possible cause because B2 is not producing data. The probability of cable failure would be higher if B2 was not expected to be offline during normal operation.

FGCU Solar Field Health Management System

Sensor Fault Detector

<table>
<thead>
<tr>
<th>Overall Sensor Health</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datalogger</td>
</tr>
<tr>
<td>--Faulty Reading</td>
</tr>
<tr>
<td>Horizontal Pyranometer</td>
</tr>
<tr>
<td>WXT 520 - Anemometer</td>
</tr>
<tr>
<td>WXT 520 - Wind Vane</td>
</tr>
<tr>
<td>WXT 520 - Ambient Temperature</td>
</tr>
<tr>
<td>AB7 Load Meter</td>
</tr>
<tr>
<td>Pad A Inverter</td>
</tr>
<tr>
<td>Pad A Generation Meter</td>
</tr>
<tr>
<td>Pad B1 Inverter</td>
</tr>
<tr>
<td>Pad B1 Generation Meter</td>
</tr>
<tr>
<td>--Faulty Reading</td>
</tr>
<tr>
<td>Pad B2 Inverter</td>
</tr>
<tr>
<td>Pad B2 Generation Meter</td>
</tr>
</tbody>
</table>

Figure 16: Main User Interface Screen Illustrating Two Faulty Readings Detected.
6 Software Testing

6.1 Application of the Handbook

The Software Testing stage is a process in software development, which receives input from the Software Implementation stage, and produces output to the Customer (the only Stakeholder in this project). The current section describes the test cases for all components of the Fineman software: the client (separately for the Connectivity Component and the Reasoning Component) and both tools (Extraction Tool and Injection Tool).

6.2 Testing the Connectivity Component

A test plan for the Fineman client’s Connectivity Component system should support the following objectives:

- Detail the activities required to prepare for and conduct the test.
- Communicate to all responsibilities the tasks that they are to perform and the schedule to be followed in performing the tasks.
- Define the test tools and the environment needed to conduct the system test.

This test plan covers a full test of the Connectivity Component software in a Windows environment. The Connectivity Component’s purpose is to log into the website http://www.solarems.net and download a .CSV file with relevant information about the solar field’s performance and current status. The software must satisfy the requirements already put forth in this document. Given the inputs of login username and password, the software outputs a file in CSV format.

Test results are presented in table 2. All tests were performed with no adverse effects witnessed. The GUI/Functionality works as intended. The software has the ability to run for extended periods of time. The performance is roughly what the designers expected (32KB when idle initially, 75KB after first fetch and there on). The JAR file also works as intended. Testing was unable to be performed with traditional tools, but it is almost unnecessary to do that. Supplying
the software with a username and password is all that is needed to test the result. A valid username and password allows the software to get the needed file; an invalid username/password combination will not allow the software to access the website and will allow the user to try again with a different username/password combination.

Table 2. Test Results for the Connectivity Component.

<table>
<thead>
<tr>
<th>Task</th>
<th>Predecessor Tasks</th>
<th>Responsibility</th>
<th>Time Spent (hrs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepare test plan</td>
<td>Complete SRS</td>
<td>Justin Hodnett</td>
<td>2</td>
</tr>
<tr>
<td>Test GUI/Functionality</td>
<td>Complete Build of Software</td>
<td>Justin Hodnett</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chris Steiner</td>
<td>20</td>
</tr>
<tr>
<td>Test Ability to Run for Extended Periods of Time</td>
<td>Complete Build of Software</td>
<td>Chris Steiner</td>
<td>96</td>
</tr>
<tr>
<td>Test Performance</td>
<td>Complete Build of Software</td>
<td>Justin Hodnett</td>
<td>1</td>
</tr>
<tr>
<td>Test JAR</td>
<td>Complete Build of Software</td>
<td>Justin Hodnett</td>
<td>1</td>
</tr>
</tbody>
</table>

6.3 Testing the Reasoning Component

Testing of the Reasoning Component has been designed to meet respective requirements 5 through 8.

6.3.1 Testing Configuration Parser (Requirement 5)

Reach: Everything

6.3.1.1 Absence of a configuration file

Temporarily remove the configuration file

The configuration parser should abort, and the software should exit.

6.3.1.2 Malformed configuration file

Supply a configuration file that contains invalid characters for at least one instrument

The software should attempt to continue parsing unaffected parts of the file (if any exist), and report a parse failure.

6.3.1.3 Cyclic dependencies

1. Supply a configuration file containing an instrument that depends on at least one of its own measurements.
2. Supply a configuration file containing an instrument with a dependency path that leads back to itself.
In both cases, parsing should continue, but the offending instrument should be flagged invalid.

6.3.1.4 Unresolvable instrument measurement relations
1. Supply a configuration file containing an instrument that references an instrument that does not exist.
2. Supply a configuration file containing an instrument that references another valid instrument, but references a measurement that does not belong to referenced instrument.
In both cases, parsing should continue, but the offending instrument should be flagged invalid.

6.3.2 Testing User Interface (Requirement 6)
Reach: N/A

6.3.2.1 No instruments have faulty readings
Supply a dataset with all measurements within operational limits
All instruments on the main user interface screen should report a status of “OK” or “Offline”

6.3.2.2 Instrument has more than one faulty reading
Supply a dataset containing at least two faulty readings for the same instrument
For each faulty reading, the software should:
1. Display a single expandable user interface element that reports the type of fault and its measured value
2. Upon user input to expand the item, list the fault’s probable causes.

6.3.2.3 Faulty reading in an instrument that measures environmental conditions
Assign a faulty reading to one of the weather sensors
The list of probable causes should not reference any other instrument in the system

6.3.2.4 Detailed instrument view
The user interface should display:
1. All manufacturer information defined in the software’s configuration file, including a working link to manufacturer specifications.
2. All of its measured values and a representation of the validity of each measurement
3. A link to return to the main user interface screen
6.3.2.5 User interface links
For each instrument displayed on the main user interface screen, traverse all of its linked screens
1. All links should be valid
2. No screen should contain a link to itself
3. For each link on the main screen, there must exist a cycle (a path of links that leads back to the
   main screen).

6.3.3 Testing Bayesian Belief Network (Requirement 7)
Reach: Post-Processor, User Interface
6.3.3.1 Configuration in which the Netica BBN’s case file cannot be loaded
Temporarily remove the BBN case file
The software should skip BBN analysis and statistical post-processing, and report a Belief
Network failure on the main user interface screen.

6.3.3.2 Network’s predicted probabilities with all input data uninitialized
Run the software with no input data multiple times
For each modeled node, the software should generate the same believed state on each run.

6.3.3.3 Network’s ability to detect, report and cope with data contradicting model behavior
Supply a measurement that is impossible given the state of all other measured data.
The software should flag the associated instrument as invali

6.3.4 Testing Offline Subsystem Tolerance (Requirement 8)
Reach: Belief Network, Post-Processor, User Interface
6.3.4.1 Test a subsystem that is allowed to be offline
Pad B2 is offline seasonally, test four datasets:
1. All measurements for the pad B2 inverter and meters are uninitialized
   The software should recognize B2 as offline
2. All measurements for the pad B2 inverter and meters have data
   The software should recognize B2 as operational
3. The pad B2 inverter has no measurements, but the meter does
   The software should recognize inverter B2 as failing, and meter B2 as operational
4. The pad B2 inverter has measurements, but the meter does not
The software should recognize inverter B2 as operational and meter B2 as failing
For cases 2-4, the instruments should be the subject of further analysis by the software.

**6.3.4.2 Test a subsystem that is not allowed to be offline**
Provide a dataset containing uninitialized values for all measurements on meter A:
The software should flag meter A as failing

**6.4 Test Plan for the Extraction Tool**

The contents of this test plan are as follows:

A. Test GUI.
B. Test CSV parsing logic.
C. Test correctness of the CSV output file.
D. Appendix of C# code for unit tests that have been completed.

**Section A - GUI Test Plan:**
1. Ensure all icons, buttons, etc. are displayed correctly.
2. Ensure all icons, buttons, etc. are meaningful and unambiguous in what their respective functions are.
3. Ensure all Action & Event listeners handling GUI operations function properly.

**Section B - CSV Parsing Logic Test Plan:**
1. Create Unit Tests for CsvParser and CsvStream classes.
2. Run Unit Tests with various control .csv files.
3. Correct CSVParser and CSVStreamText classes’ logic as needed to ensure accurate results.
4. Run Tests with actual downloaded file from Christopher Steiner and Justin Hodnett.

**Section C - .CSV File correctness Test Plan:**
1. Test successful creation of .csv file of extracted sensor data.
2. Ensure data is formatted correctly.
3. Test and verify data is accurate and complete.
6.5 Testing the Injection Tool

Test Case 1: Sensor selection drop-down menu
- Test Objective: Ensure that when the sensor selection drop-down menu is clicked, all the available sensors are listed.
- Test Description:
  1. Start GUI program.
  2. Click on sensor selection drop-down menu
- Expected Results: The list of available sensors shall be displayed in the sensor selection drop down menu.

Test Case 2: Standard sensor range display
- Test Objective: Ensure that when a sensor is selected, the standard sensor range is displayed on the left side of the GUI.
- Test Description:
  1. Start GUI program.
  2. Click on sensor selection drop-down menu
  3. Click on sensor in drop-down menu
- Expected Results: A display of the sensor’s standard ranges shall be displayed on the left-hand side of the screen when a sensor is selected via the sensor menu.

Test Case 3: Numerical fault value input box
- Test Objective: Ensure that when the user clicks on the fault value input box and inputs a numerical value via the keyboard, the value is displayed in the fault value input box.
- Test Description:
  1. Start GUI program.
  2. Click on fault value input box
  3. Enter a numerical value via keyboard
- Expected Results: The fault value input box shall receive and display numerical, user input.
Test Case 4: Fault value variance input box

- Test Objective: Ensure that when the user clicks on the fault value variance input box and inputs a numerical value via the keyboard, the value is displayed in the fault value variance input box.
- Test Description:
  1. Start GUI program.
  2. Click on fault value variance input box
  3. Enter a numerical value via keyboard
- Expected Results: The fault value variance box shall receive and display numerical input.

Test Case 5: Frequency of injection drop-down menu

- Test Objective: Ensure that when the frequency of injection drop-down menu is clicked, a list of time durations is displayed in the frequency of injection drop-down menu.
- Test Description:
  1. Start GUI program.
  2. Click on frequency of injection drop-down menu
- Expected Results: The list of time durations shall be displayed in the frequency of injection drop-down menu.

Test Case 6: Save Injection Parameters Button

- Test Objective: Ensure that when the save injection parameters button is clicked, the current fault parameters are saved to a text file.
- Test Description:
  1. Start GUI program.
  2. Change fault parameters
  3. Click the save injection parameters button
- Expected Results: The fault parameters shall be saved to a text file.

Test Case 7: Load Injection Parameters Button

- Test Objective: Ensure that when the load injection parameters button is clicked, a fault parameters file is loaded and displayed.
• Test Description:
  1. Start GUI program.
  2. Click the load injection parameters button
  3. Click file to load
• Expected Results: The fault parameters shall be loaded from a text file and data displayed in correct fields.

Test Case 8: Start Injection Button
• Test Objective: Ensure that when the start injection button is clicked, faults are injected into the csv file.
• Test Description:
  1. Start GUI program.
  2. Change fault parameters
  3. Click the start injection button
• Expected Results: Faults shall be injected into the csv file according to the current fault parameters.

Test Case 9: Terminate Injection Button
• Test Objective: Ensure that when the terminate injection button is clicked, the program stops injecting faults.
• Test Description:
  1. Start GUI program.  2. Change fault parameters
  3. Click the start injection button  4. Click the terminate injection button
• Expected Results: The program shall stop injecting faults into the CSV file.

Test Case 10: Exit Button
• Test Objective: Ensure that when the load exit button is clicked, the program closes.
• Test Description:
  1. Start GUI program.
  2. Click the exit button
• Expected Results: The program shall exit.
7 Conclusion

The objective of this project, to apply guidelines of the NASA Systems Engineering Handbook to the development of software for fault detection and diagnostics in a solar power plant, has been accomplished with moderate success. The software package to retrieve data in real time from the solar plant data logger and derive conclusions on probable faults and their causes has been developed and is operational. In addition, two tools assisting in the processing of measurement data have been developed.

During the course of this project, the team has acquired extensive knowledge from various areas of system and software development, including: the design of data acquisition systems in solar plants, general fault and health management techniques, use of Bayesian belief networks as a reasoning tool, and the application of specific guidelines from the NASA Systems Engineering Handbook that can be related to software.

The project is complete in its current form, but can be expanded upon indefinitely. There are many comments that we can make on this project. The most important is that while Java has some easy solutions for logging into an HTTP website, it lacks an easy solution for logging into an HTTPS website. We ended up using a package typically used for software testing purposes (htmlunit) to finally get a reliable HTTPS login.

As with everything in Computer Science, the design of the Bayesian Belief Network is the result of a series of compromises and trade-offs. The most profound trade-off has to do with the number of measurements modeled. Because the team who developed the Bayesian Belief Network are neither experts on solar power, nor statisticians, it is impractical to develop equations to model the probability of each instrument state. Instead, the state of each instrument is modeled using estimated probabilities for each discrete combination of input values.

Future work should be done to more accurately distribute the probability of failure and to reduce the complexity associated with increasing the number of modeled measurements.
8 References


[33] Software and Systems Engineering Vocabulary. URL: http://www.computer.org/sevocab

[34] HtmlUnit Website, Gargoyle Software, August 2010, URL: http://htmlunit.sourceforge.net/


[36] Keystore Class, Oracle Corporation, Redwood City, California, 2010, URL: http://download.oracle.com/javase/1.4.2/docs/api/java/security/KeyStore.html/
Appendix 1.

local_timestamp,
utc_timestamp,
Datalogger Voltage (V),
Datalogger Temperature (°C),
POA Irradiance (W/m²),
Cell Temperature (°C),
Horiz Irradiance (W/m²),
Ambient Temperature (°C),
Wind Speed (m/s), Wind Speed Max (m/s),
Wind Direction (Deg),

COMM Status (),
AC Power (kW),
AC Voltage AN (V),
AC Voltage BN (V),
AC Voltage CN (V),
AC Current Phase B (A),
AC Current Phase C (A),
AC Current Neutral (A),
AC Current Phase A (A),
AC Energy Total (kWh),
AC Frequency (Hz),
AC Apparent Power (VA),
AC Power Factor (),
Inverter COMM Status - Pad C1 (),
Inverter AC Power - Pad C1 (kW),
Inverter DC Power - Pad C1 (kW),
Inverter DC Voltage - Pad C1 (V),
Inverter AC Voltage - Pad C1 (V),
Inverter Total DC Current - Pad C1 (A),
Inverter AC Energy Total - Pad C1 (kWh),
Inverter Fault A - Pad C1 (),
Inverter Fault B - Pad C1 (),
Inverter Run State - Pad C1 (),

COMM Status - Pad C1 (),
AC Power - Pad C1 (kW),
AC Voltage AN - Pad C1 (V),
AC Voltage BN - Pad C1 (V),
AC Voltage CN - Pad C1 (V),
AC Current Phase B - Pad C1 (A),
AC Current Phase C - Pad C1 (A),
AC Current Neutral - Pad C1 (A),
AC Current Phase A - Pad C1 (A),
AC Energy Total - Pad C1 (kWh),
AC Power Factor - Pad C1 (),
AC Frequency - Pad C1 (Hz),
AC Apparent Power - Pad C1 (VA),
Inverter COMM Status - Pad B2 Holmes ()
Inverter DC Power - Pad B2 Holmes (kW)
Inverter AC Power - Pad B2 Holmes (kW)
Inverter Total DC Current - Pad B2 Holmes (A)
Inverter DC Voltage - Pad B2 Holmes (V)
Inverter AC Voltage - Pad B2 Holmes (V)
Inverter AC Energy Total - Pad B2 Holmes (kWh)
Inverter Fault A - Pad B2 Holmes ()
Inverter Fault B - Pad B2 Holmes ()
Inverter Run State - Pad B2 Holmes ()
AC Current Phase B - Pad B2 Holmes (A)
AC Power Factor - Pad B2 Holmes ()
AC Current Phase C - Pad B2 Holmes (A)
AC Frequency - Pad B2 Holmes (Hz)
AC Current Neutral - Pad B2 Holmes (A)
AC Apparent Power - Pad B2 Holmes (VA)
AC Energy Total - Pad B2 Holmes (kWh)

COMM Status - Pad B2 Holmes ()
AC Power - Pad B2 Holmes (kW)
AC Voltage AN - Pad B2 Holmes (V)
AC Voltage BN - Pad B2 Holmes (V)
AC Voltage CN - Pad B2 Holmes (V)
AC Current Phase A - Pad B2 Holmes (A)
AC Energy Record - Pad C1 (kWh)
Draker Modeled Inverter AC Power - Pad C1 (kW)
Modeled AC Power (Base) - Pad C1 (kW)
PV Index (Base) - Pad C1 (%)
AC Energy Record - Pad B2 Holmes (kWh)
Draker Modeled Inverter AC Power - Pad B2 Holmes (kW)
Modeled AC Power (Base) - Pad B2 Holmes (kW)
PV Index (Base) - Pad B2 Holmes (%) Aggregate AC Power (kW)
Weighted Avg POA Irradiance (W/m²)
Aggregate Modeled AC Power (Base) (kW)
Aggregate PV Index (Base) (%)
import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.cookie.CookiePolicy;
import org.apache.commons.httpclient.cookie.CookieSpec;
import org.apache.commons.httpclient.methods.*;
import org.apache.commons.logging.*;
import java.io.*;

public class LoginByHttpPost {
    static final String LOGON_SITE = "www.solarems.net";
    static final int LOGON_PORT = 80;

    public LoginByHttpPost() {
        super();
    }

    public static void main(String[] args) throws Exception {
        System.setProperty("javax.net.ssl.trustStore", 
                            "/home/clearscreen/mykeystore");
        HttpClient client = new HttpClient();
        client.getHostConfiguration().setHost(LOGON_SITE, LOGON_PORT, "http");
        client.getParams().setCookiePolicy(CookiePolicy.BROWSER_COMPATIBILITY);

        GetMethod authget = new GetMethod("/projects/36-fgcu-ab7/overview");
        client.executeMethod(authget);

        System.out.println("Login form get: "+ authget.getStatusLine().toString());

        // release any connection resources used by the method
        authget.releaseConnection();

        // See if we got any cookies
        CookieSpec cookiespec = CookiePolicy.getDefaultSpec();
        Cookie[] initcookies = cookiespec.match(LOGON_SITE, LOGON_PORT, "/", 
                                               false, client.getState().getCookies());

        System.out.println("Initial set of cookies:");

        if (initcookies.length == 0) {
            System.out.println("None");
        } else {
            for (int i = 0; i < initcookies.length; i++) {
                System.out.println("- " + initcookies[i].toString());
            }
        }

        PostMethod authpost = new PostMethod("/projects/36-fgcu-ab7/overview");

        // Prepare login parameters
        NameValuePair action = new NameValuePair("action", "/user_sessions");
        NameValuePair url = new NameValuePair("url", ":");
NameValuePair userid = new NameValuePair("text", "XXXXXXXXXX@XXXXXXXXXXX");
NameValuePair password = new NameValuePair("password", "XXXXXXXXXX");
authpost.setRequestBody(new NameValuePair[] { action, url, userid, password });

client.executeMethod(authpost);

System.out.println("Login form post: "+ authpost.getStatusLine().toString());

// release any connection resources used by the method
authpost.releaseConnection();

// See if we got any cookies
// The only way of telling whether logon succeeded is
// by finding a session cookie
Cookie[] logoncookies = cookiespec.match(LOGON_SITE, LOGON_PORT, "/", false, client.getState().getCookies());

System.out.println("Logon cookies:");
if (logoncookies.length == 0) {
    System.out.println("None");
} else {
    for (int i = 0; i < logoncookies.length; i++) {
        System.out.println("- " + logoncookies[i].toString());
    }
}

// Usually a successful form-based login results in a redirect to
// another url
int statuscode = authpost.getStatusCode();
if ((statuscode == HttpStatus.SC_MOVED_TEMPORARILY)
    || (statuscode == HttpStatus.SC_MOVED_PERMANENTLY)
    || (statuscode == HttpStatus.SC_SEE_OTHER)
    || (statuscode == HttpStatus.SC_TEMPORARY_REDIRECT)) {
    Header header = authpost.getResponseHeader("location");
    if (header != null) {
        String newuri = header.getValue();
        if ((newuri == null) || (newuri.equals(""))) { newuri = "/"; }

        System.out.println("Redirect target: " + newuri);
       GetMethod redirect = new GetMethod(newuri);
       client.executeMethod(redirect);
       System.out.println("Redirect: "+ redirect.getStatusLine().toString());
       InputStream inputStream = redirect.getResponseBodyAsStream();
       InputStreamReader isr = new InputStreamReader(inputStream);
    }
}
BufferedReader in = new BufferedReader(isr);

String inputLine;

while ((inputLine = in.readLine()) != null) {
    System.out.println(inputLine);
}

in.close();

// release any connection resources used by the method
redirect.releaseConnection();

} else {
    System.out.println("Invalid redirect");
    System.exit(1);
}
}