# C S 373 Software Engineering
# World Crisis Report

Team: Import Antigravity
Harrison He, Jerome Martinez, Michael Pace
Joe Peacock, Andy Hsu, Justin Salazar

Summer 2012

## 1 Executive Summary

This technical report, by group Import Antigravity, provides a step-by-step analysis of the class group project; the construction of a web application displaying data on relevant world crises using Python and the Google App Engine (GAE). This database must be capable of importing, merging, and exporting data to create dynamic pages, and the website must implement an intuitive user interface in order to increase awareness about the status of crises, related organizations, and related persons around the world.

Methods of analysis include an introduction to the problem at hand, the thought process of our design to handle the problem, the implementation of that design, and the in-depth testing and evaluation of the implementation. In the introduction, we detail certain topics such as the problem, any interesting discoveries related to the problem, any limitations on our part, as well as our approach to solve it. We delve into our design choices as well, exploring our designs in XML, UML, and the UI, and show our implementation of the datastore, framework, and the structure of our website as a solution to the stated problem.

We finish with our methods using the GAEUnit test suite to create an expansive testbed in order to evaluate and enforce that our methods work as intended.

Any and all diagrams relevant to the analysis can be found within their respective section inside the technical report. Results of the data analyzed within this report show the structure of the project created by our group as a whole and our progress across different phases. This technical report, detailing the World Crisis project, may not be complete as the project is a work in progress and will be updated as necessary by any group member.

# 2 Introduction

## 2.1 What is the problem?

We were tasked to construct a website backed by a Google App Engine datastore. The website must be capable of creating dynamic HTML pages, and the server-side logic must be written in Python. This website must be able to import, merge, and export XML files in order to populate the datastore, and must provide an intuitive interface for the user to navigate.

## 2.2 Why is this interesting and important?

This project requires us to work in a group with others and provides us with a chance to work with tools and utilities we would otherwise never touch upon. This project marks a milestone in our education, giving us real world experiences in a controlled environment before we graduate.

Creating a website using the GAE allows the opportunity to dynamically populate the site contents using data extracted from formatted XML files. The

import functionality of the site as well as a unified schema allows for a painless compilation of unique data from different sources. Using this model also makes managing content updates simpler as well.

## 2.3 Why is this hard?

Among the challenges this project provided for our group were: learning to use the GAE, creating an XML instance/schema that wasnt overly generic and provided regularity amongst the groups, designing the user interface, designing models for the datastore, and writing functions to effectively to import and export XML data. Another unexpected difficulty involved adapting a testing suite whose support has been discontinued since 2009 (certain functionalities in GAEUnit, such as creating an isolated test datastore, appear to no longer work correctly with the current version of GAE). Overcoming these difficulties has proved to be greatly beneficial and exciting to our group in terms of learning and experiences.

## 2.4 Key Components of our Approach

Our approach to the problem in this project involved constructing various handlers, each providing a specific functionality, to give utility to the database as a whole. The project is broken up into multiple modules: one for importing, another for exporting, another for defining models in the datastore, and so on. Additionally, we created several templates to allow dynamic rendering of html pages based on input data from the main handler.

## 2.5 Specific Limitations

One of the largest limitations when creating the World Crisis Database was the utilization of the Google App Engine. Our inexperience with this utility limited

our ability to create the most efficient code possible. We took time to familiarize ourselves with the new system, which could have been spent on more unit tests, adding site functionality, or improving performance.

Another limitation presented to the team was the rigidity of the XML schema. Because the use of XML is required, users not familiar with the language would not be able to easily import information about World Crises into the database. This is especially the case since the imported XML file must validate with the specific schema the GAE database identifies with. This particular limitation seems to suggest a potential new feature to be added: an Instance Builder tool. Whether we will have time to implement this in the final stage remains to be seen.

This group project introduced many languages and tools (GAE, GAEUnit, django, XML, ElementTree, etc.) which were new to us. This resulted in us being hindered by a lack of experience and knowledge of these tools and prohibited us from using them to their full potential. This limitation, however, became less significant throughout over the course of the project. Ultimately, we are required to make several large unplanned investments of work time, in order to become proficient with new tools and frameworks.

# 3 Design

## 3.1 Design Descisions

Most of our design decisions were made with simplicity for the user in mind. Whether that user was the design team (as it was when we devised flat and easily accessible datastore models), or an end user smoothly browsing our website,

simplicity was our intention.

## 3.2 Project Diagrams

Figure 2 shows a high level diagram representing our project. Our project consists of three main divisions: all the data populated by XML instances or the GAE datastore, code that organizes and restructures the data, and the human interface that specifies how to display the data.

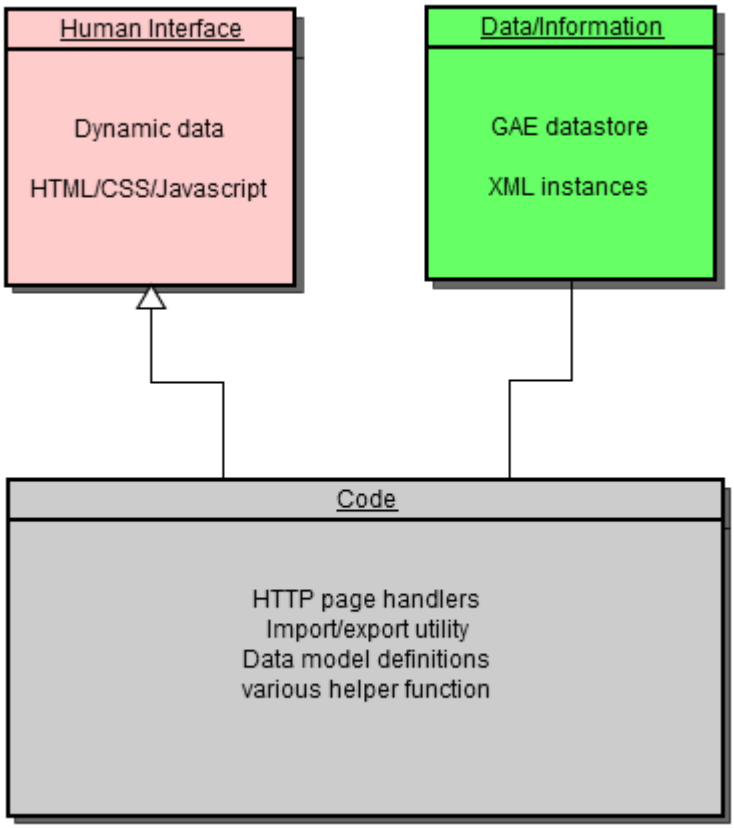


Figure 1: Overall layout of our project

## 3.3 UI

The user interface was designed with the average user in mind. We aimed to keep navigation of the site simple, with links to splash pages in the same place on every page. At the same time, we hoped to engage the user by providing smooth transitions between pages. Our assortment of images on the homepage provides both an aesthetic appeal and guidance for the user. Each image corresponds to one data model object in the datastore. Upon hovering over the main links (crisis, people, organizations), the homepage provides an animated filter showing only the pertinent images.

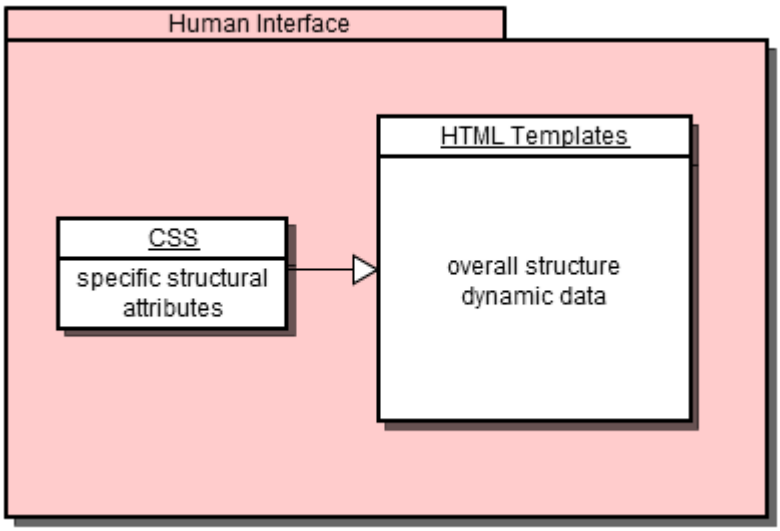


Figure 2: The basic UI component for our project

### 3.3.1 Data and Information

All the data is provided solely by the users in the form of XML instances. The data held in the XML instances are temporary. The data flows from the XML instances into the Code and parses all the data. From there, all the parsed data

is stored in the GAE datastore where the data is persistent.

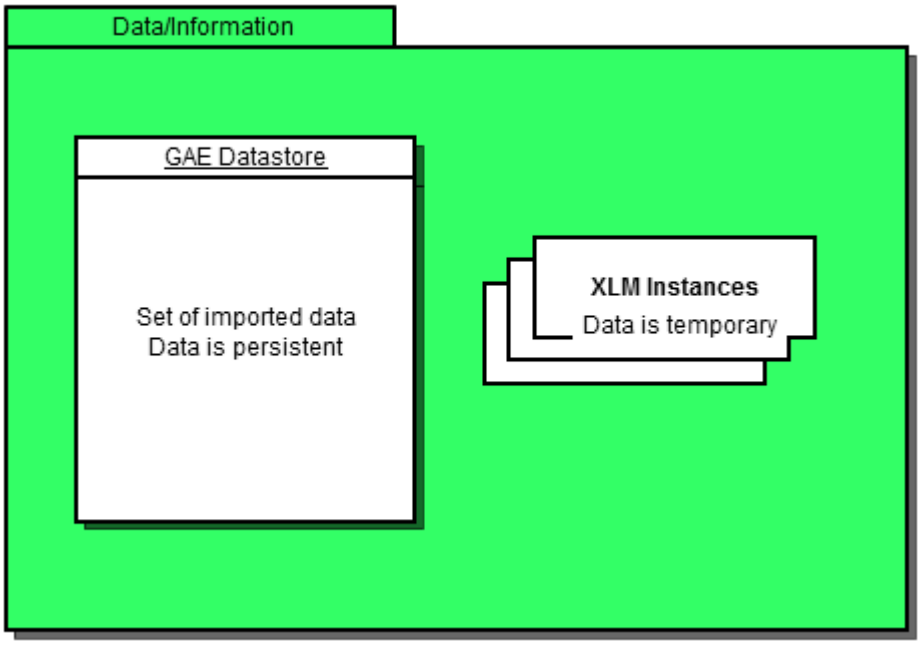


Figure 3: The data and information of our project is stored in the form of data models in the GAE Datastore and XML files

## 3.4 Data/Information

We designed the the GAE datastore to be as flat as possible to allow easy access to any piece of data we may require. This served two purposes. First, it was a precaution in case our schema was not chosen after the first phase. Since our models were not closely tied to our schema, we wouldnt have had to make significant changes to them in the case that a different schema was chosen. Second, it became fairly simple to dynamically build any html page using Django templates, as well as rebuilding an ElementTree in order to export as an XML file. We chose to split the information into four different classes:

one for crises, one for organizations, one for people, and one for external links. This made it easy to differentiate them within the code. We matched types in the models to corresponding types in the schema. For some fields, such as the info_history field in the Crisis model, we chose to use a db.TextProperty(), since db.StringProperty() has a max limit of 500 characters.

### 3.5 Special Features Added

During XML import, we ping every linked image in the instance were working with to ensure its URL isnt dead. If the ping returns a valid HTTP header, we add the URL to the datastore. Otherwise we leave the URL field blank and, later during page display, filter out blank URLs using JavaScript. This way no broken images will provide an unpleasant browsing experience to the end user. We noticed a number of websites from previous semesters are littered with broken images, so we aimed to future-proof our site by filtering them out. Unfortunately, this feature isnt free: it increases import time, but this is a minor and unavoidable drawback. In order to expedite testing, our current import page provides a check-box that allows the user to disable URl checking. This will most likely not be included in the final project.

Another special feature is the ability to import not using a local XML file but using a URL with an XML format so long as it validates against the schema. This allows our implementation to pull data off the export pages of other groups using the same schema. While this functionality is outside the requirements of the project, we thought it would be an extremely useful addition, as it saves the user from having to export and save a new XML file every time a groups instance data changes.

### 3.6 UML

See figure 1 for our UML diagram. Because our models are stored in a flat structure in the GAE datastore. The UML of our data models represent the same flat structure. None of the data models have any relationships.

### 3.7 XML

Our goal was to create a schema that was general enough to accommodate many types of data but granular enough to allow access to specific pieces of data. We chose to use many complex types and nest them throughout other complex types in order to prevent redundancy and allow for modularity in how the schema could be altered with minimal impact on the rest of the code. We further improved the schema by sharing it with the class in a public Google document to allow collaboration, facilitate discussion, and nudge us all toward consensus.

## 4 Implementation

### 4.1 Solution?

The problem tasked to us was solved using a wide variety of tools which allowed us to create a robust website backed by a datastore. We used jquery, isotope.js, and Bootstrap to provide a seamless and engaging front-end interface for users to navigate. XML parsing using Elementtree and minixsv along with Django template-enabled dynamic html pages aided in importing other XML files and depicting such data without the use of redundant coding.

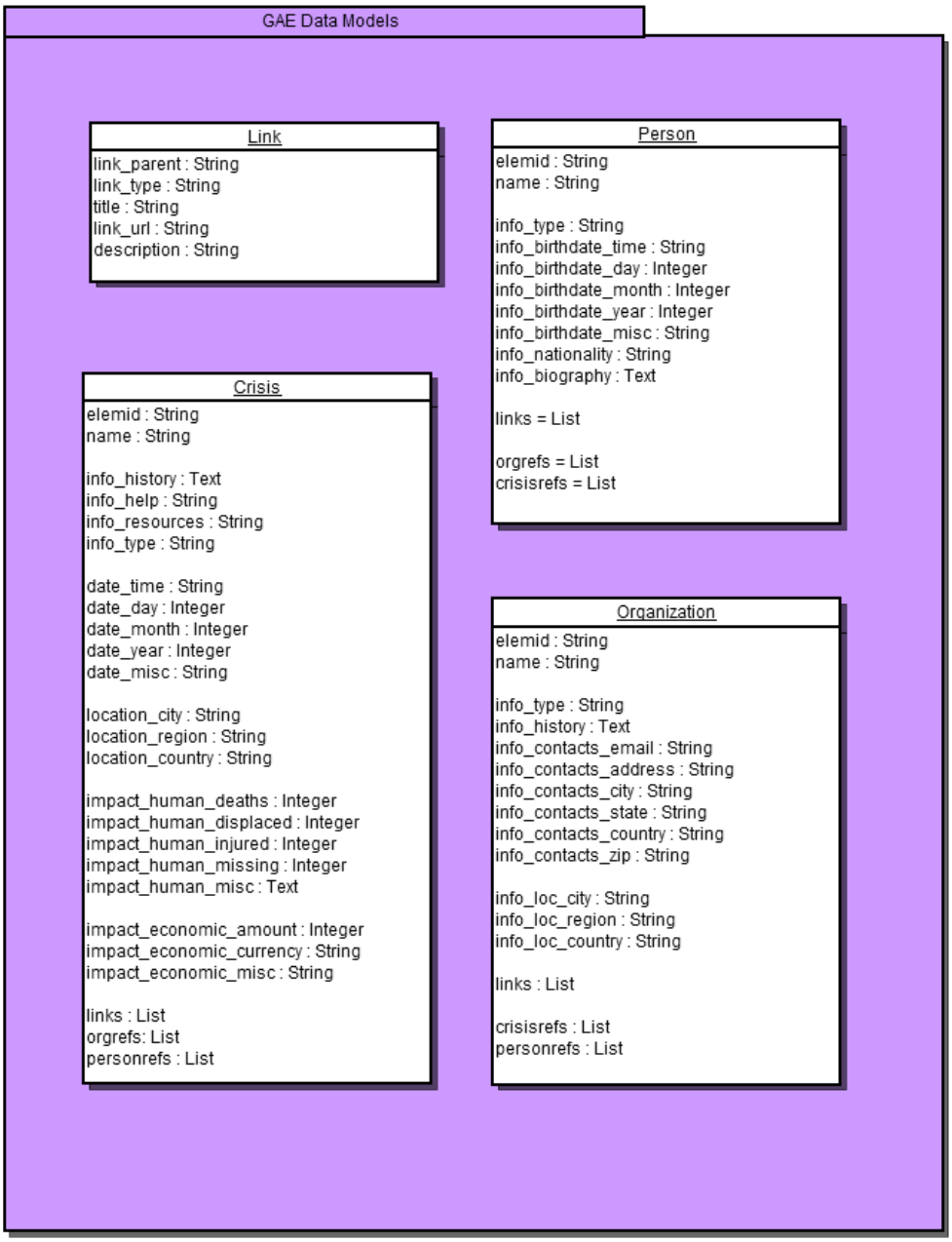


Figure 4: Overall layout of our project

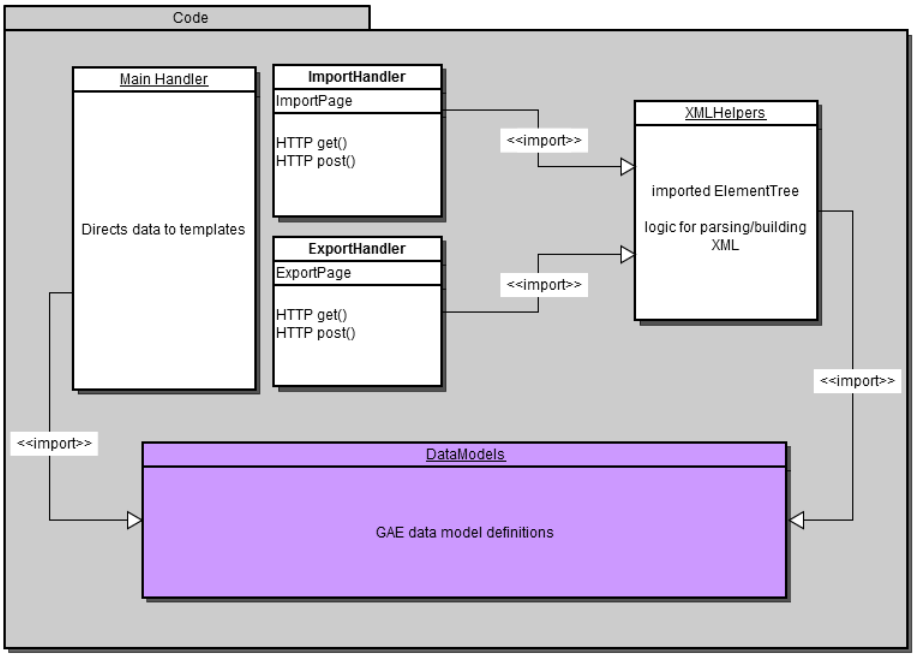


Figure 5: At a high level, this diagram describes the organization of our code

## 4.2 Frameworks and tools

### 4.2.1 JQuery and JavaScript

Jquery is a JavaScript library that facilitates common in-browser tasks like ensuring cross-browser compatibility and navigating and manipulating the DOM tree. We used it extensively: it provides the fading for page transitions, for example, and also the ability to use isotope.js.

Isotope.js is a jquery plugin that made possible the animations on our websites home page. Upon hovering over a link to a category – i.e., Crises, People, or Organizations – images on the page are filtered, so that only those pertaining to that category remain. Implementing this feature, however, required a bit of a workaround due to a lack of foresight when designing the datastore models.

The images that display on the page are stored in the Link models. Links have a link_parent property which contains the ID of the relevant issue (e.g., haiti), but not a property describing the type of the parent (i.e., crisis). This is an essential piece of knowledge to be able to filter images as we want. Our solution is to build a JavaScript object which serves as a lookup table. It contains three properties: crises, organizations, and people. Each of those properties is associated with an array containing link_parent names. For instance, the crises property has four elements this phase: wafrica, haiti, iraq, and chernobyl. With this knowledge, we are able to filter the images according to type.

Bootstrap is a set of .html, .css, and .js files that aim to serve as a starting-off point for new websites. We made use of the .css files, and then wrote our own styling on top of that. Our site benefits from the baseline design provided by Bootstrap, but is set apart by our own additions and edits to that design.

#### 4.2.2 Django Templates

Django temples were necessary in order to display all the data stored in the GAE datastore. These templates allowed for html files to be dynamically populated by data passed in by related Python code (MainHandler.py). This functionality allowed us to avoid creating numerous static html files for each topic, in favor of three templates which could be applied to any data in the datastore. Further, embedded python code in the html template allows for certain logic to be considered when displaying data. Examples of this include humanization of numbers (1000 -¿ 1,000) and displaying missing data as Unknown.

#### 4.2.3 minixsv

The minixsv package was imported into the database in order to implement XML instance to schema validation. This package was chosen because it is both

lightweight and written in pure Python, which was necessary for it to interact with the GAE. Using the parseAndValidateXmlInputString() function from the minixsv library, the database is able take in an instance, verify it against a specified schema, and raise an exception when the instance is invalid. The function validXML() utilizes this function call in order to return a simple True or False value.

## 4.3 Datastore

When viewing the datastore, the most important elements to understand are the unique Models that are being stored. For the World Crisis Database, four different models are defined: Link, Crisis, Organization, and Person.

The Crisis, Organization, and Person models store data for each respective member of the database. Some examples of unique fields in each are human deaths in the Crisis model, contact information for the Organization model, and date of birth for the Person model. In the XML instance each of these fields are built into a tree structure, with certain fields being descendants of a certain type. An example of this is that both deaths and injuries fall into the category of Human Impact. Maintaining the tree structure when utilizing models, however, quickly becomes tedious and confusing. For this reason, we opted for these models to remain flat. This means that instead of having attributes descend from others, all attributes are contained in simple variables (strings and ints) at the same level of prominence in the Model. This remains true for all attributes in the Crisis, Organization, and Person models except for the links attribute in each.

The links attribute utilizes ListProperty(db.Key) in order to create a list

of references to another model, Link. The Link model is another flat model that stores information about any external link that is added to the database. The links attribute in Crisis, Organization, and Person needs to be a mutable attribute because it is not known exactly how many links will be uploaded to the database for a specific model.

## 4.4 Import/Export

At the end of phase 2, the import functionality works as expected. It takes the given XML and verifies against the chosen schema using the minixsv tool detailed in the implementation tools. It then overwrites any data already within the datastore if the XML verifies, and rejects the XML if it does not and retains the existing datastore. An additional method using a URL is detailed within the special feature added section.

The import functionality is handled by ImportHandler.py which checks the XML given against the schema using minixsv, and then calls parseXML() within XMLHelpers.py. parseXML() parses the XML and stores data as an ElementTree into the GAE datastore. At this point in the project, merge functionality is not supported.

The export functionality is called from ExportHandler.py with buildXML() and builds multiple subtrees for every crisis, organization, and person. For each one, buildCrisis(), buildOrganization(), or buildPerson() are called from within XMLHelpers.py, and appended onto an ElementTree by calling ElementTree.SubElement(). Then the entire ElementTree is returned using the tostring() method, resulting in a well-formatted XML file for internet browsers.

## 4.5 Files and directory structure

The files for this project were split into multiple modules, to make the GAEUnit testing much easier to accomplish. Most files are in the /wc directory, with tests being in /wc/tests, .css files in /wc/stylesheets, .js files in /wc/js, and documentation files in /wc/docs.

## 4.6 Task Delegation

- Repository integrity : Jerome Martinez

- Main site hosting and maintenance : Michael Pace

The site is hosted on the GAE servers, and maintained using the GAE executable.

- UI design and interaction : Michael Pace and Justin Salazar

The UI is implemented using HTML, CSS, and JavaScript. We aimed to keep it simple and elegant.

- XML schema construction and verification : Harrison He and Andy Hsu

The XML schema defines the structure and data types of an XML document. In the World Crisis database, the schema was defined to contain specific information about a crisis, organization, or person. The structure was chosen to have 3 main complexTypes: Crisis, Organization, and Person. These, in turn, are wrapped by a worldCrises complexType. Any instance that is imported into the datastore is first verified against the schema, and is parsed into the datastore if it passes.

- UML : Jerome Martinez

- Import/Export : Joe Peacock and Michael Pace

Import and export handlers are split into two separate modules, both of which make use of the functions in the XMLHelpers module. Imports are handled by uploading a file through an html form, and then passing it as a python file object to ElementTree.Parse(). In the case of the import from URL special feature, we use urllib2.urlopen() to create a file-like object containing the raw data found at that URL. Next, the generated ElementTree is broken down into individual sub-trees through use of the find()/findall() functions. These subtrees are then used to construct instances of our db.Model classes. Conversely, exports are handled by querying the datastore, constructing an ElementTree from the returned model objects, and then calling its toString() method. The export string is then written to the page in proper XML format.

- Documentation : Joe Peacock

Handler classes, models, and helper functions are commented where necessary and pydoc-generated .html files are included with the source code. The most troublesome part of this was actually getting pydoc to work with all the modules specific to GAE.

- GAEUnit testing : Group Collaboration

Extensive unit test were created in order to make sure that all method calls are fail-safe. Regular testing and corner-case testing helps to assure correct functionality of the program.

-Technical report : Group Collaboration

# 5 Evaluation

## 5.1 Testing our solution

Our implementation was tested extensively using the GAEUnit test suite on top of our own structured testing in order to allow for a flawless user experience in navigating our database websites.

## 5.2 Unit tests

We tested each of our methods and functions within the handler classes and XMLHelpers module with at least 3 unique unit tests each, for a total of 35 tests. This allowed us to verify that the program was parsing all information from the imported XML files as expected, the datastore was receiving the information reliably and was non-volatile, and html pages could be built consistently.

## 5.3 Perfomance Tetsts

At this current phase in the project, we are currently not implementing any performance testing. We have various other tests to ensure the project works as expected but we are programming with efficiency in mind.

# 6 Conclusion

In conclusion, we believe we have fulfilled all the uniform requirements of this assignment as of phase 2, as well as designed exceptional code to provide a

smooth experience both for the programmer and user. This was accomplished by giving a simplistic design to the back-end code running the website database, and focusing on providing an intuitive experience to the user, with a minimal, but still interesting level of interaction with the data.