Grocery Shopping Assistance for the Visually Impaired

ENG 100, Fall 2008

University of California, San Diego

Team Members

Jerry Ni
Amalia Prada
Marissa Sasak
Alvin Cabrera
Hourieh Fakourfar

Advisor

Kai Wang
PhD Candidate
Computer Science Department

Serge Belongie
Assistance Professor
Computer Science Department

Community Client

John Miller
NFB Representative

Sponsors:

Table of Contents

1. Introduction 1
1.1 Executive Summary 1
2. Soylent Grid 2
3. Approach and Methodology 3
4. Tasks and Sub-teams 3
4.1 User Interface (UI) 3
4.2 Database Team 5
4.3 Integration of the User Interface and Database Code 9
4.4 Population of Database 9
4.4.1 A Brief Description of the XML File 10
4.4.2 Python Script 10
4.4.3 SQLAlchemy: Object Relational Mapper Toolkit 11
5. Tools and Technologies 13
6. Future Work 14
7. Acknowledgement 14
8. References 15
9. Appendix 15

1. Introduction

1.1 Executive Summary

There are currently 1.3 million legally blind people living in the United States who face daily obstacles with routine tasks, especially in regards to their experiences within supermarkets and stores. Developing assistive technologies and handheld devices allows for the possibility of increasing independence for blinds and visually impaired. Currently, many grocery stores treat those that are blind as “high cost” customers, and dramatically undersell to this market, neglecting to take their needs into consideration. The use of computational vision can be advantageous in helping these blind customers, as restrictions such as the limited ability of guide dogs, frequently changing store layouts, and existing resources do not allow for a completely independent shopping experience. Using technologies such as object recognition, sign reading, and text-to-speech notification could allow for a greater autonomous solution to the growing problem.

In conjunction with Calit2, UCSD’s Computer Vision Lab, and TIES, the GroZi project is working to develop a portable handheld device that can “see”, helping the visually impaired navigate more efficiently within difficult environments as well as better locate objects and locations of interest. GrloZi’s primary research is focused on the development of a navigational feedback device that combines a mobile visual object recognition system with haptic feedback. Although still in its early stages of development, when complete, the GroZi system will allow a shopper to navigate the supermarket, find a specific aisle, read aisle labels, and use the handheld MoZi box to then scan the aisle for objects that look like products on the shopper’s list (compiled online and downloaded onto the handheld device prior to going into the store).

This quarter, under the supervision of our advisor, Kai Wang, we pursuit the computer vision aspects of the project that allows for autonomous detection and localization in the near future. Thereby, our team successfully customized the User Interface (UI) for new labeling tasks as well as created a computer program that allows for inserting and storing data into the database as effortlessly as possible. However, due to time constraint, there is still room for improvement. This means that the incoming contributors to this project should continue on improving our codes that could further improve the outcome of the project as a whole. The following document will serve as a description of what we have accomplished thus far, what we have learned and overcome, and the processes involved in designing and implementing a usable and accessible interface for the blind to assist future members of TIES GroZi team.

2. Soylent Grid

The Soylent Grid, a subcomponent of GroZi, combines Human Computational Tasks (HCTs) and Machine Computable Tasks (MCTs) by labeling images and solving common vision problems such as segmentation and recognition.1 For GroZi to work efficiently, a database, containing images of grocery store products and their appropriate labels, is needed. Soylent Grid functions by sending these images to a computer which performs an MCT algorithm in order to identify any text labels. This system performs optical character recognition, also known as OCR, on product packaging in order to differentiate between certain grocery items as peanut butter and grapes. Because these items contain differing colors, fonts, and background images, recognizing the product's text is challenging. From a Soylent Grid perspective, the main goal is to obtain differing product images and to obtain such information as the product's brand name and a description of the item. If the computer algorithm cannot successfully perform this task, the image is then sent to users who identify the location and the content of the text (HCTs).

In addition to labeling grocery store products via text, Soylent Grid also functions as a way to secure websites. These Soylent Grid images can be used as a portal before accessing private information such as a bank account or an email address. To protect such personal information, these images can be combined with reCAPTCHA, in which users must enter the text displayed in a distorted image in order to obtain the desired information. In this way, Soylent Grid provides a double service, a win-win situation, by not only labeling images for the GroZi database, but also by providing security for users regarding any personal information.

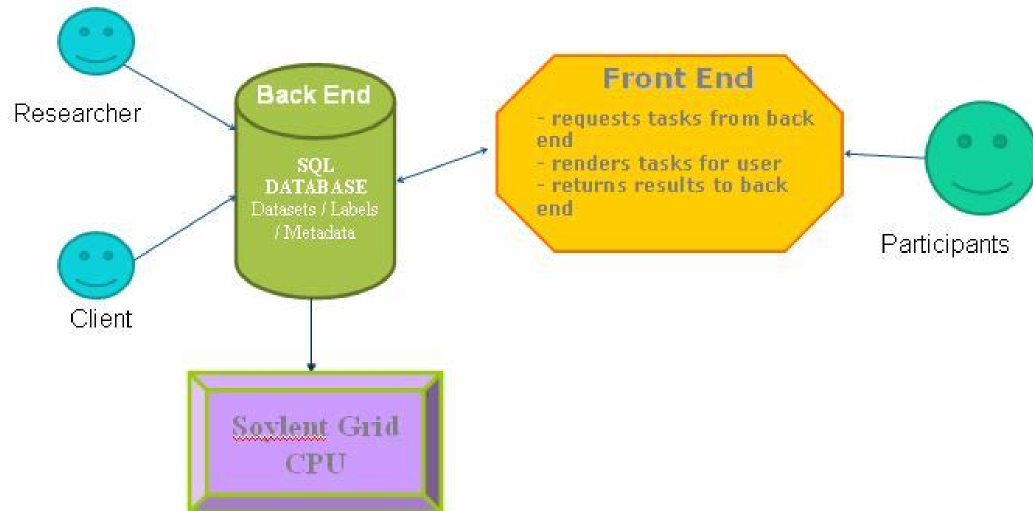

Figure 1. Illustrates the structures of Soylent Grid

3. Approach and Methodology

In order to design a grocery shopping assistant for the visually impaired, we need a program that will allow us to save images of grocery store items in a real environment. To avoid the intese human labor, we exploited the concept introduced above, Soylent Grid. In order to proceed, we then split into three groups this quarter to set up the necessary preliminary steps:

- **User Interface :** Amalia Prada & Alvin Cabrera
    - Customizing User Interface for new labeling tasks and storing results into a database for use in future computer vision systems.
- **Database:** Jerry Ni & Marissa Sasak
    - The database team's task consisted of adding a table to the Soylent Grid code in order to store labels to a database.
- **Population of Database:** Hourieh Fakourfar
    - Develop a computer program to populate the database

4. Tasks and Sub-teams

4.1 User Interface

The main goal for this quarter was to work on the User Interface of Soylent Grid, in which users are supposed to identify and label the database images. To program the interface, Java was used with a sophisticated text editor, Eclipse, because it not only compiles and runs the script efficiently, but also accesses the libraries more easily. In addition to this, Google Web Toolkit was used to build the JavaScript and to thereby convert the code into front-end applications in the Java programming language.

Figure 2. Output of TyT_Task.java

To improve the User Interface of the Soylent Grid, two scripts were used: TyT_Task.java and WiW_Task.java. The TyT_Task.java creates a picture and a text box, as displayed in Figure 1, and asks the user to input the text displayed in the image. This task allows items to be categorized by product brand names and descriptions. Similarly, the WiW_Task.java displays a picture with a toolbox and allows the user to draw a rectangular box around the desired text.

Both codes from TyT_Task.java and WiW_Task.java were combined in order to produce a final code that displays a picture and prompts the user to input text and use the toolbox in order to draw a rectangular outline around any related text. An example of this code's output is demonstrated with Figure 2, in which the user inputs "Cup Noodles Chicken Flavor" into the textbox and uses the Rectangle Tool to outline the two character strings.

Figure 3. Example of output from WiW_Task.java (Before and After Input)

WiW_task.java. The first modification involved altering Line 89, so that the text on the first button changed from "Erase" to "Textbox":

```
final Button erase = new Button("TextBox");
```

Additionally, code from TyT_Task.java was added to Line 81 in order to produce a textbox that prompts the user to input descriptions of the image:

```
//make a textbox
final TextBox tb = new TextBox();
String tbText;
//(tbText = tb.getText(); this line is moved to line 108)
```

In this code, the final Textbox function creates the textbox in which users input descriptions of the image. The user's input then becomes accessed with the method tb.getText() and eventually outputted into a file titled 00000.log.

Furthermore, to create the displayed Toolbar Buttons to the right of the image, new code was added to Line 172:

```
HorizontalPanel buttonPanel = new HorizontalPanel();
        buttonPanel.add(erase);
        buttonPanel.add(save);
        buttonPanel.add(generate);
        buttonPanel.add(tb);
```

In this code, the first command, HorizontalPanel, defines buttonPanel and its various buttons of Erase, Save, and Generate.

When modifying the code, the function, tbText = tb.getText() was moved from Line 84 to line 108, inside the OnClick function. Because this produced an error, the variable was renamed to 's', as displayed below:

```
//Save the coords
final Button save = new Button("Save");
        save.addClickListener(new ClickListener()
        {
        public void onClick(Widget sender)
        {
        String s = tb.getText();
        saveData(curImage,listener.getData());
        //listener.displayData();
```

As depicted in Figure 3, the function listener.clr() was uncommented on Line 94.

```
81              //make a textbox
82              final TextBox tb = new TextBox();
83              String tbText;
84              tbText = tb.getText();
85              //System.out.println(tbText);
86
87
88              // Clear the picture
89              final Button erase = new Button("TextBox");
90                  erase.addClickListener(new ClickListener()
91                  {
92                    public void onClick(Widget sender)
93                    {
94                        System.out.println("here");
95                     // listener.clr();
96                    }
97                  }
98                  );
99                  erase.setWidth("120px");
```

Figure 4. Snippet of WiW_Task.java Code for Generating Textbox

Because of the introduction of a new ‘s’ variable, another variable had to be defined in the function “SaveData” on Line 319:

public void saveData(String image, String s, ArrayList data)

All other functions referencing this new variable also needed this ‘s’ variable in order to function properly. As a result, ‘s’ was inserted into the saveData function on Line 109:

saveData(curImage, s, listener.getData());

Finally, because an ‘s’ variable was defined, the String variable tbText was no longer needed and thereby deleted from all areas of the code.

4.2 Database Team

The database task this quarter consisted of adding to and changing the preexisting Soylent Grid code. This was a challenging task because we did not have previous experience using JAVA, My SQL, etc. We also made use of Google Web Toolkit and Eclipse.

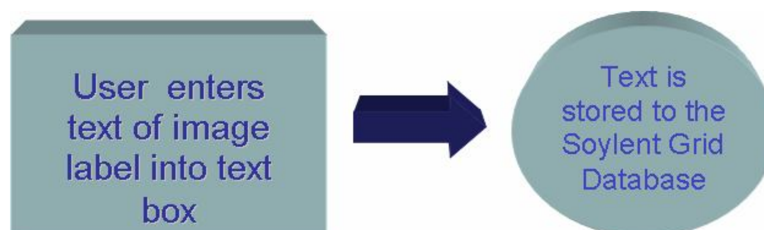

Figure 5. Illustrates the design flow used by the database team

Since the project requires a large amount of data stored in a place that can be acessed universally by different machines, a database was created though MySQL to store all of the data relating to the Soylent Grid project. Eclipse was used to edit the Soylent Grid code and compile everything together using the programming language JAVA. Google Web Toolkit provided a list of pre-made classes that were used as a front-end to our web application logic. It interfaces the source code that runs the Soylent Grid into a format that can be used online similar to HTML. Java is a programming language that is similar to C therefore using it gives a more powerful platform to getting where we want.

The following code was added to initialize the table *SoylentGrid* once. We ran it in a separate java program, Eclipse, so that our table would not get rewritten each time we ran the *Soylent Grid* program. It establishes a connection with the SQL server and creates a table *SoylentGrid* with automatically incremented rows and data categorized by the columns. First column is a 40 character string for the location of the actual image. Second column is a 40 character string of what the user inputs into the text box.

```
import java.sql.*;

public class Connect
{
public static void main (String[] args)
{
Connection conn = null;

try
{
String userName = "root";
String password = "root";
String url = "jdbc:mysql://localhost/test";
Class.forName ("com.mysql.jdbc.Driver").newInstance ();
conn = DriverManager.getConnection (url, userName, password);
System.out.println ("Database connection established");
Statement s = conn.createStatement ();
int count;
s.executeUpdate ("DROP TABLE IF EXISTS SoylentGrid");
s.executeUpdate (
"CREATE TABLE SoylentGrid ("
+ "id INT UNSIGNED NOT NULL AUTO_INCREMENT,"
+ "PRIMARY KEY (id),"
+ "location CHAR(40), name CHAR(40))");
}
catch (Exception e)
{
System.err.println ("Cannot connect to database server");
}
```

```
finally
{
if (conn != null)
{
try
{
conn.close ();
System.out.println ("Database connection
terminated");
}
catch (Exception e) { /* ignore close errors */ }
}
}
}
}
```

The following code was inserted into filetoolimpl.java of the existing *SoylentGrid* code under the "type this" task. The code initializes the connection with the SQL server and updates the table *SoylentGrid* whenever the "save" button for the "type this" task is executed. It saves the data into its respective columns and updates the table. We then call the data back from the table to verify that it is being written in.

```
String userName = "root";
String password = "root";
String url = "jdbc:mysql://localhost/test";
Class.forName ("com.mysql.jdbc.Driver").newInstance ();
conn= DriverManager.getConnection (url, userName, password);
System.out.println ("Database connection established");
Statement s = conn.createStatement ();

String STMT = "INSERT INTO SoylentGrid(location, name)" +
"VALUES" +
"('" + image + "','"+ data + " ')";
System.out.println("Check: " + STMT);
s.executeUpdate(STMT);
s.executeQuery ("SELECT id, location, name FROM SoylentGrid");
ResultSet rs = s.getResultSet ();
int count = 0;
while (rs.next ())
{
int idVal = rs.getInt ("id");
String nameVal = rs.getString ("location");
String catVal = rs.getString ("name");
System.out.println (
"id = " + idVal
```

```
+ ", location = " + nameVal
+ ", name = " + catVal);
++count;
}
rs.close ();
s.close ();
System.out.println (count + " rows were retrieved");
```

Several of the lines in the included code are SQL commands such as "INSERT INTO SoylentGrid(location, name)" this line enters value into the SoylentGrid table that was implemented on the SQL database. Further information abuot these commands can be found in a tutorial that was very helpful in our groups task http://www.kitebird.com/articles/jdbc.html.

One of the most integral parts in the integration between the UI and the Database was recognizing the functions of each code. The database had made modifications to the writeFile_TYT for updating the table in the database for the "type this" task, however the UI team had made modifications for the "where's waldo" task. A simple change from option 3 to option 2 in the taskcreator.java from "type this" to "where's waldo." And the following code in the TyT_task.java yielded the integration between the hard work accomplished by both teams this quarter. Currently not all information is being saved correctly since the function is calling a different number of variables than what is originally defined by the writeFile_TyT function, however this is an easy fix. Due to time constraints this quarter, we were unable to finish this but can easily be a starting point for the next.

```
if (GWT.getModuleBaseURL().substring(7, 21).equals("localhost:8888"))
    {
      if(url.equals("null"))
        saver.writeFile_TyT("00000000.log", curImage, data, callback);
      else
        saver.writeFile_TyT("log/"+name+".log", curImage, data, callback);
    }
    else
      if(url.equals("null"))
        saver.writeFile_TyT(path+"/00000000.log",
                curImage,
                data,
                callback);
      else
        saver.writeFile_TyT(path+"/log/"+name+".log",
                curImage,
                data,
                callback);
```

4.3 Integration of the User Interface and Database Code

After both teams successfully completed both aspects of the project (User Interface and Database), the codes were then combined into WiW_Task.java. This new code, which displays an image, allows the user to enter text labels into the textbox and to draw a rectangle around any related information. When the "Save" button is then selected, the integrated software adds this text label to the SQL database. If the user exits the GWT program, contents of the SQL database is printed to the ECLIPSE console. At this moment, however, the software is unable to save the rectangle's coordinates for reference or future use. Saving both these coordinates and the text label in the SQL database is a possible future design task.

4.4 Population of Database

In the past, graduate students along with other contributors to this project, created a database that consists of one hundred and twenty images from grocery store items. This quarter, the team of undergraduate students focused more on expanding the database so it can hold as many as 30,000 images or more. However, this is not a simple task because each image that is being inserted to the database needs to be correctly labeled. In the past, this was done manually by graduate students and volunteers. Hence to replace this tedious method with a computer based strategy, Soylent Grid has been introduced to the team as an effective solution that allows for effortless way of labeling the images as they are being inserted to the database. However, before one proceeds to exploit the full capability of Soylent Grid, there are some preliminary steps that are needed to be considered. One of the preliminary steps was to write a program that allows for populating the database. This part carries a major load because it essentially links the front end (and the back end. For this task to be achievable, three major parts needed to be integrated: 1) XML label file that holds all the labels and information regarding images, 2) Pythonic program that defines the XML tags in terms of objects, 3) SQLAlchemy Object Relation Mapper that includes dialects for MySQL and has nice features such as function-based query construction that supports SQL clause to be implemented via Python functions and expressions. Figure 6 is a block diagram that provides a quick and easy description of the design flow as we have envisioned.

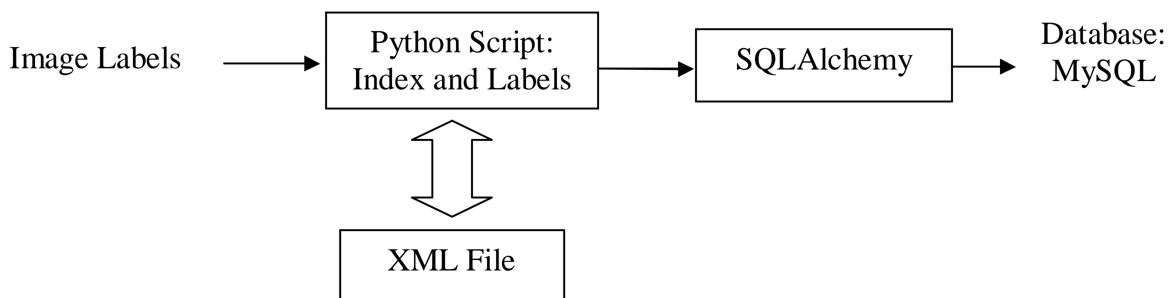

Figure 6: A block diagram explaining the design flow of the population of database

The block diagram above shows the steps necessary to accomplish this task. Each of these stages will be thoroughly explained in the following subsections.

4.4.1 A Brief Description of the XML File

The XML (labels0.xml) is simply a text ground truth on one or more images from each GroZi product that Kai Wang, a computer science PhD candidate, and the TIES team collected. Figure 7 is a screen shot that provides one with a brief overview of the content of this file.

```
- <grozi>
  - <image>
      <imageName>1/web/JPEG/web1.jpg</imageName>
    - <taggedRectangles>
      - <taggedRectangle x="23" y="51" height="20" width="31">
          <tag>ReNU</tag>
        </taggedRectangle>
      </taggedRectangles>
    </image>
  - <image>
      <imageName>1/web/JPEG/web2.jpg</imageName>
    - <taggedRectangles>
      - <taggedRectangle x="156" y="168" height="9" width="47">
          <tag>SOLUTION</tag>
        </taggedRectangle>
```

Figure 7. Illustrates the content of the XML File

As it is illustrated in Figure 7, each XML file has a root, <grozi> and as many as 120 children, <image>. Each image tage has its own children in the following branchers: image name and taggedRectangles. Each taggedRectangle can have several taggedRectangle with different image coordinates and height and widths.

4.4.2 Python Scrip

For this part, one of our team members, Hourieh Fakourfar, has written a short python program that allows for defining each of the XML nodes as a pythonic object. In essence, this program parses the pre-existing XML file which outputs a pythonic version of the XML file (i.e. that is an object orientated conversion of the XML file). The reason for this program is to allow for easy indexing, inserting, and omitting certain labels or tags in pursuit of expanding our database to be able to hold as many as 30,000 or more images. Furthermore, since we chose SQLAlchemy as an arbitrary program to connect to the database and query our data into the database, this short python script comes handy as a starting point. However, this code still needs further modification. For instance, one could write a few lines of codes to save the output into a file or write a schema for it.

To briefly explain how this code functions, each XML tag was defined as a separate class. Each class has a few functions serving different functionalities. For

instance, each class has an add, delete, and a check for redundancy. To give you a better understanding, the code for one of the classes is provided as a visual aid, which illustrates some of the functions within each class.

```
class Image():
    def __init__(self, imageName):
        self.imageName = imageName
        self.taggedRectangles = []
    def _str_(self):

str1="<image>\n\t<imageName>"+self.imageName+"</imageName>\n\t<taggedRe
ctangles>\n"
        strBuild=""
        str2="\t</taggedRectangles>\n</image>"
        for tr in self.taggedRectangles:

strBuild+=tr._str_().replace("<t","\t\t<t").replace("</taggedRectangle"
,"\t\t</taggedRectangle")+"\n"
        return str1+strBuild+str2

    def equals(self,nameTag):
        return self.imageName == nameTag.imageName

    def addTaggedRectangle(self, taggedRectangle):
        for i,v in enumerate(self.taggedRectangles):
            if v.equals(taggedRectangle):
                return
        self.taggedRectangles.append(taggedRectangle)

    def removeTaggedRectangle(self, taggedRectangle):
        for i,v in enumerate(self.taggedRectangles):
            if v.equals(taggedRectangle):
                self.taggedRectangles.remove(i)
                return

    def getTaggedRectangle(self, index):
       return self.taggedRectangles[index]

    def taggedRectanglesLen(self):
       return len(self.taggedRectangles)
```

As it appears above, most of these functions are self-explanatory. The last two functions, however, needs a brief clarification. The two functions of getTaggedRectangles and taggedRectanglesLen are very important for flexible indexing and labeling. Please note that you can learn more about each of these functions and their purpose in Python tutorial provided at:

http://www.python.org/doc/2.5.2/tut/tut.html

4.4.3 SQLAlchemy: Object Relational Mapper Toolkit

SQLAlchemy is an object relational Mapper Toolkit that was chosen arbitrarily as one of many useful toolkits that helps us to populate the database. One of the reasons to choose this program among others was the fact that it includes a feature called function-based query construction. This feature allows one to define SQL clauses to be built through Python functions and expressions. Another remarkable feature is raw SQL statement mapping. This feature enables the programmer to accommodate raw SQL statements in conjunction with plain result sets. Thereby, object instances can be generated from these results in the same approach as any other ORM operation [3].

Using this special toolkit, we successfully generated a program that allows one to easily populate the database. In this section, we will briefly explain the process by which we generated this program.

Prior to programming, one needs to first complete all of the required program installation. Figure 8 is a visual illustration of the necessary steps for necessary installations.

Figure 8. Steps for installation [10]

Requirements:

Eclipse 3.3 (python requires only the platform release, and jython also requires JDT)
Python 2.3.x or 2.4.x
Java 1.4 or higher

After installation, one needs to configure the interpreter via Eclipse. To further configure the interpreter, one should follow the steps below:

- 1. Go to: window > preferences > pydev > Interpreter - (Jython/Python).
- 2. Choose the interpreter you have installed in your computer (such as python.exe or jython.jar)
- 3. Select the paths that will be in your **SYSTEM PYTHONPATH**. It's important to select only those folders that will **not** be used as the root for some project of yours.

After those steps, you should have a screen as presented below:

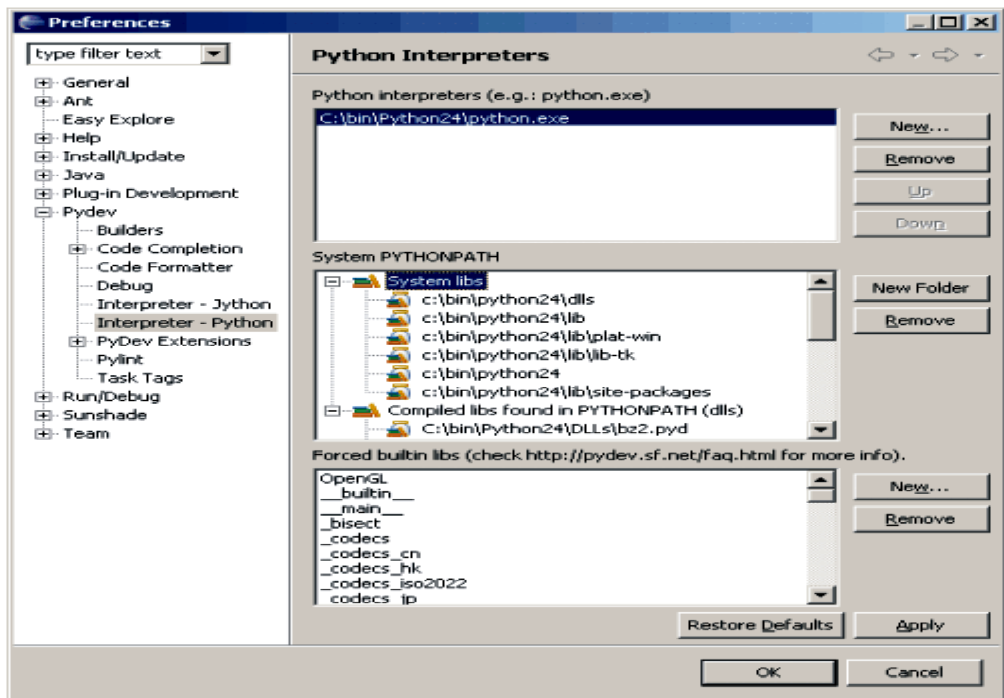

Figure 8. Steps of Configuration [10]

After all of the installations and configurations, one can proceed with the online tutorial of SQLAlchemy. This online tutorial provides helpful insights on how one can

adequately take advantage of this toolkit in order to populate the database as efficiently as possible.

In order to enable the SQLAlchemy functionality and use them in our code, we need to:

1. Eclipse: import sqlalchemy: Accessing SQLAlchemy module
2. Install MySQL
3. Go to: Start > run > cmd > mysql –u root –p
4. Enter your password (if you have designated one)
5. Create a table:
    - create sg
    - if you have created one already: use sg
6. Then on Eclipse import a class called create_engine which is responsible for managing the connection to the database:
    - from sqlalchemy import create_engine
    - engine = create_engine('mysql://root:12345@localhost/sg')

After completing these steps, one should be able read the code was written this quarter and make appropriate changes to the code.

5. Tools and Technologies

This quarter, we have learned and exploited the following tools and technologies which made our implementations and developments feasible.

1. Toolkit
    a. Google Web Toolkit
    b. SQLAlchemy
2. Programming Languages
    a. Java
    b. Python
3. IDE
    a. Eclipse
4. Database
    a. MySQL

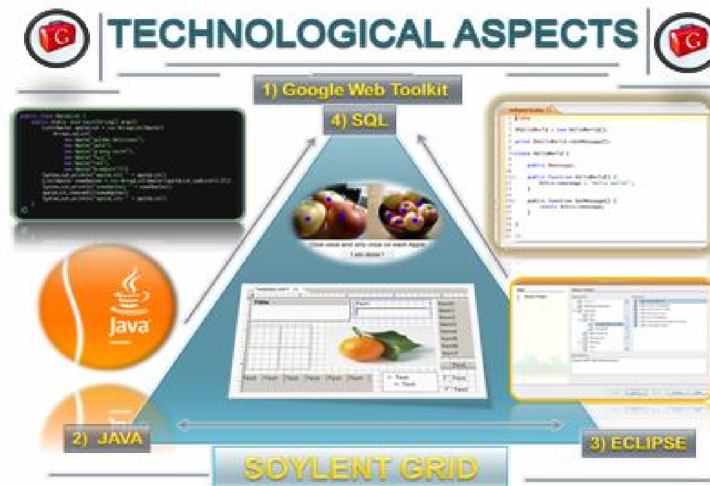

Figure 9. Illustrates the main technologies and tools that was used for software development

6. Future work

Future work for this specific project involves modifying the table and code to all Soylent Grid tasks. For this quarter we only implemented the table for the "type this" task which involves a product image and a query that asks the user to type the name of the product into the text box. Other soylent grid tasks require more intricate implementation. For the task that requires the user to draw boxes around the name of an item would need to use a larger table. Since product names are of varying lengths, the table would not be able to be categorized the same way not to mention each letter of the word would require 4 distinct coordinates from the square. When it is saved into the database, the team would need to figure out how to categorize the table with either flexible rows or columns that would accommodate different objects and to keep the four coordinates paired to each letter. For new Grozi TIES members, it is important to gain background knowledge on coding as soon as possible and to familiarize oneself with the Soylent Grid code. Also since the finished code combining the User Interface and Database does not successfully save text labels and rectangular coordinates, a possible future design task includes solving this problem by altering or adding to the WiW_task.java code.

Future possible tasks are outlined in the e-mails sent from John Miller and Kai Wang (Refer to Appendix for corresponding e-mails). Miller suggested deleting Line 83 (String tbText) because the variable was already defined, but not referenced. This task has been completed and has resulted in a cleaner code with no warnings. Miller also suggested improving the overall design of the User Interface code so that the erase button clears the text from the textbox. In particular, he recommended adding Line 94 before listener.clr() so instructions would be added to tb.setText(" "). This task has also been completed. When the "Erase" button is selected, the text box becomes redrawn with no text inside. Also, the WiW_Task.java code needs to include the instruction tb.setText(" ").

7. Acknowledgments

We all would like to acknowledge the vast help and support provided by our advisor, Kai Wang, and community client, John Miller. Without their constant mentorship this project would not be feasible what-so-ever.

8. References

1. S. Steinbach, V. Rabaud, and S. Belongie. (2008 December 11) Soylent Grid: It's Made of People!
2. README.txt Kai Wang
3. SQLAlchem (Features): http://www.sqlalchemy.org/features.html
4. SQLAlchemy(Tutorial): http://www.sqlalchemy.org/docs/05/ormtutorial.html
5. Python: http://www.python.org/doc/2.5.2/tut/tut.html
6. SQL: http://www.w3schools.com/sql/default.asp
7. Wiki on Object Relational Mapper: http://en.wikipedia.org/wiki/Object-relational_mapping
8. Java: http://java.sun.com/docs/books/tutorial/
9. GWT: http://www.softwaredesign.co.uk/gwt.html
10. Eclipse (python interpreter): http://pydev.sourceforge.net/
11. Eclipse (Downloads): http://www.eclipse.org/downloads/

9. APPENDIX

A.1 E-mails Regarding Possible Future Design Tasks

1) From Kai's email:

Regarding number 3: I just tried it, and nothing gets written to the file when SAVE is pressed without a shape drawn out, although the message "data saved" still pops up. It would be nice to only allow the button to be clicked when both text is in the box AND a rectangle is specified in the image.

I did a search in the java documentation and found you can do something like this:

```
void        setEnabled(boolean b) //Enables (or disables) the button.
```

Here's an example of how the code could look:

```
Button b = new Button("save");
b.setEnabled(false);
```

---

that will create a button that the user can see but won't be able to click on; it will probably have a faded gray appearance. Something like that would be cool.

HTH,
Kai

2) From John's email:
On Mon, Dec 8, 2008 at 10:14 AM, John Miller <j8miller@soe.ucsd.edu> wrote:
> Hello,
>
> I have reviewed the UI code sent me by Amalia from last Wednesday's
> development session.
>
> Here are some questions and areas for improvement
>
> Line 83: String tbText; this variable is defined (originally for debugging
> or part of development) but is not referenced
>
> Action: remove this line, the code will read cleaner and no warning will be
> generated. **.- DONE**
>
> 2. This is more about the overall design of the UI code.
>
> If a textbox is filled in, then the erase button is pressed, then does the
> text get cleared from the textbox?
>
> Should it?
>
> If so, I suggest in the execution code for erase button
>
> Line 94 before listener.clr() to add instruction tb.setText(""); a parallel
> action is done in tyt_task.java for erase.
>
> 3. In wiw_task.java code, is it possible to select the save button with no
> rectangle formed or with no text typed in?
>
> For the text in particular, here the text string s would contain nothing or
> s == NULL.
>
> What is the correct action to take here? It seems NULL text should not be
> submitted to the database.
>
>
>
> Kai – would you comment on the action for task 3?

A.2 How to Set-up and Install Soylent Grid

Below is shown the process to download Eclipse, the GWT libraries, and to set up the Soylent Grid.

1. Download and unzip these libraries somewhere:

- Eclipse
- GWT: unzip to a directory, example: [C:\lib\gwt]
- GWT Widgets: http://sourceforge.net/project/showfiles.php?group_id=169692&package_id=193560&release_id=604758
  place into a directory, example: [C:\lib]
- SoylentGrid_dist: unzip to a directory, example: [C:\code\dist]

2. Set up the SoylentGrid project

- Create a new directory [C:\code\SoylentGrid]
- Open Command Prompt and run these commands in the newly created directory
  > C:\lib\gwt\projectCreator -eclipse SoylentGrid
  > C:\lib\gwt\applicationCreator -eclipse SoylentGrid com.ucsd.client.SoylentGrid
- Copy the starter code to the correct locations in the newly created project
  - in the new project directory, delete the folders named 'src' and 'test'
  - copy the folders 'src', 'test', 'Pictures', and 'www' from [C:\code\dist\] into the new project directory [C:\code\SoylentGrid]
  - copy the file 00000000.log from [C:\code\dist\00000000.log] to the new project directory [C:\code\SoylentGrid\00000000.log]

3. Open SoylentGrid in Eclipse and run!

- Create new Java Project
  - Create project from existing source, Browse to the directory where your project is: [C:\code\SoylentGrid]
  - Name it 'SoylentGrid'
  - Finish.
- Add the widgets library to your classpath
  - Right click on the SoylentGrid project in the left panel -> Properties
  - Click on Java Build Path on the left panel
    - Add External JARs
    - Select the widgets JAR you downloaded earlier: [somewhere in C:\lib]

- Run it
  - Run -> Run Configurations
    - Click 'Java Application'
    - Click 'New Launch Configuration'
    - Name: 'SoylentGrid'
    - Project: 'SoylentGrid'
    - Main class: 'com.google.gwt.dev.GWTShell'
    - Click the 'Arguments Tab'
      - Program Arguments: '-out www com.ucsd.SoylentGrid/SoylentGrid.html'
      - VM Arguments: '-Xmx256M'

- Wait for the GWT Browser to read something like: “Soylent Grid: Welcome to the creating SG link web page...”
- Click Redirect
- You should see a picture of some product with a text box and some buttons -- if not, there is a problem.
- Enter some text and click Save
- A dialog should pop up saying the data was saved.
- Open up the file: [C:\code\SoylentGrid\0000000.log] and check if the word you typed was saved or not -- if not, there is a problem.
- If there are no problems up to this point, you are set!

Once we have everything downloaded we can start working with Java scripts. We start getting familiar with Java, we followed a tutorial on line: http://www.javabeginner.com/jinternalframe.htm
We created a window or frame of work, with a picture, a text box and a couple of buttons by copying and modifying the examples of the tutorial.

A.3 Key Features of SQLAlchemy

Some of the key features at a glance:

Supported Databases	SQLAlchemy includes dialects for SQLite, Postgres, MySQL, Oracle, MS-SQL, Firebird, MaxDB, MS Access, Sybase and Informix; IBM has also released a DB2 driver. The corresponding DB-API 2.0 implementation (or sometimes one of several available) is required to use each particular database.
Unit Of Work	The Unit Of Work system, a central part of SQLAlchemy's Object Relational Mapper (ORM), organizes pending create/insert/update/delete operations into queues and flushes them all in one batch. To accomplish this it performs a topological "dependency sort" of all modified items in the queue so as to honor foreign key constraints, and groups redundant statements together where they can sometimes be batched even further. This produces the maxiumum efficiency and transaction safety, and minimizes chances of deadlocks. Modeled after Fowler's "Unit of Work" pattern as well as Hibernate, Java's leading object-relational mapper.
Function-based query construction	Function-based query construction allows SQL clauses to be built via Python functions and expressions. The full range of what's possible includes boolean expressions, operators, functions, table aliases, selectable subqueries, create/update/insert/delete statements, correlated updates, correlated `EXISTS` clauses, `UNION` clauses, inner and outer joins, bind parameters, and free mixing of literal text within expressions. Constructed expressions are compilable specific to any number of vendor database implementations (such as PostGres or Oracle), as determined by the combination of a `Dialect` and `Compiler`

	provided by the implementation.
Separate mapping and class design	Database mapping and class design are totally separate. Persisted objects have no subclassing requirement (other than 'object') and are POPO's : plain old Python objects. They retain serializability (pickling) for usage in various caching systems and session objects. SQLAlchemy "decorates" classes with non-intrusive property accessors to automatically log object creates and modifications with the `UnitOfWork` engine, to lazyload related data, as well as to track attribute change histories.
Eager-loading of objects	Whole graphs of related objects can often be loaded with a single query that is automatically generated to join the appropriate tables together, known as *eager loading*. The alternative to eager loading, *lazy loading*, loads related objects via distinct query executions. Each type of loading produces identical results and are interchangeable, allowing configuration at any level as well as query-time selection of the relationship-loading method to be used.
Composite (multiple-column) primary keys	In SQLAlchemy, primary and foreign keys are represented as sets of columns; truly composite behavior is implemented from the ground up. The ORM has industrial strength support for meaningful (non-surrogate) primary keys, including mutability and compatibility with ON UPDATE CASCADE, as well as explicit support for other common composite PK patterns such as "association" objects (many-to-many relationships with extra meaning attached to each association).
Self-referential tables and mappers	Self-referential tables and mappers are supported. Adjacency list structures can be created, saved, and deleted with proper cascading, with no code overhead beyond that of non-self-referential structures. Self-referential structures of any depth can be eagerly loaded in a single statement using joins; you set how deep you'd like to go.
Inheritance Mapping	Explicit support is available for single-table, concrete-table, and joined table inheritance. Polymorphic loading (that is, a query that returns objects of multple descendant types) is supported for all three styles. The loading of each may be optimized such that only one round trip is used to fully load a polymorphic result set.
Raw SQL statment mapping	SQLA's data mapper can accomodate raw SQL statements as well as plain result sets, and object instances can be generated from these results in the same manner as any other ORM operation. Any hyper-optimized query that you or your DBA can cook up, you can run in SQLAlchemy, and as long as it returns the expected columns within a rowset, you can get your objects from it. Statements which represent multiple kinds of objects can be used as well, with results received as named-tuples, or with dependent objects routed into collections on parent objects.
Pre- and post-processing of data	The type system allows pre- and post- processing of data, both at the bind parameter and the result set level. User-defined types can be freely mixed with built-in types. Generic types as well as SQL-specific types

are available.