

StoryGrab

IB COMPUTER SCIENCE HL

INTERNAL ASSESSMENT

By Calvin Kinateder, 2019

Criterion A: Planning

The problem I am tackling for my IA is for the ComOps department at AEP. ComOps deals with the buying and selling of power. They needed a way to find news about the price of power and really all things that would interest that department on consumer sites. The goal of my program will be to get a list of webpages that could be potentially relevant to a certain keyword. Google does not do this for them because Google a) does not do the best job of sorting by relevance in this instance, and b) does not write to a database. The client’s name is Joe Sheridan and he is in the ComOps department. A family member of mine has worked with him on a few occasions and that’s how I got in contact with him.

Rationale for Solution

My Java program will search a list of sources for a given keyword and write the output to a database sorted by relevance. The sources will include most consumer news sites and will also include a Twitter API as well as the option to add more sources. Pre-installed sources will include

* <http://bbc.com/>
* <http://foxnews.com/>
* <http://www.huffingtonpost.com/>
* <http://www.nbcnews.com/>
* <http://www.nytimes.com/>
* <http://www.wsj.com/>
* <http://www.usatoday.com/>
* <http://news.google.com/>
* <http://www.rollcall.com>
* <http://www.latimes.com>
* <http://www.wired.com>
* <http://www.cnn.com>
* <http://www.npr.org>
* <http://abcnews.go.com/>
* <https://www.usnews.com/news>
* <https://www.yahoo.com/news/>
* <https://www.ap.org/en-us/>

In order to read these sources, I’ll need to use a third-party library to connect to the webpage and save the data to a field in the program. I’ll also need to design a searching algorithm to write to some sort of List storing the increasing number of “hits.” I plan on making the GUI fully functioning at all times, so I will need to use Thread to stop the program from hanging while the sources are being searched. Finally, the user will need to possess some way of reading the output, possibly through a web browser or, as Mr. Sheridan would also like, a database.

I decided to implement the solution in Java 8 because:

* Fairly easy to create GUI with
* Object-oriented language so my multi-class design fits well
* Easy-to-follow syntax
* Free to download and distribute

Success Criteria

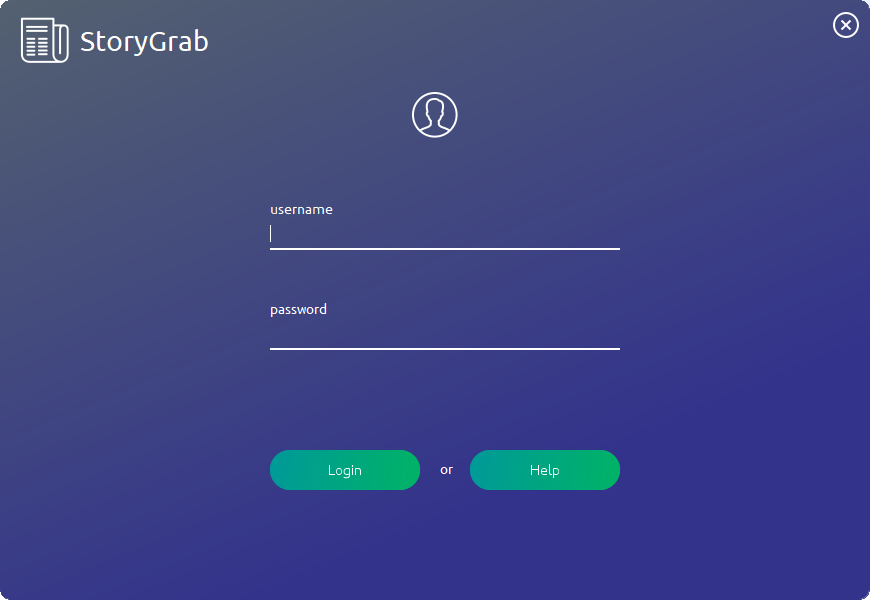
* Must search for the given keyword
* Results must be at least reasonable
* Execution time should be low
* Program must be efficient (multithreading)
* Interface must be intuitive and easy to use
* Output must be easy to interpret and easy to find
* Code must be readable with comments
* Interface should look modern

Criterion B: Solution Overview

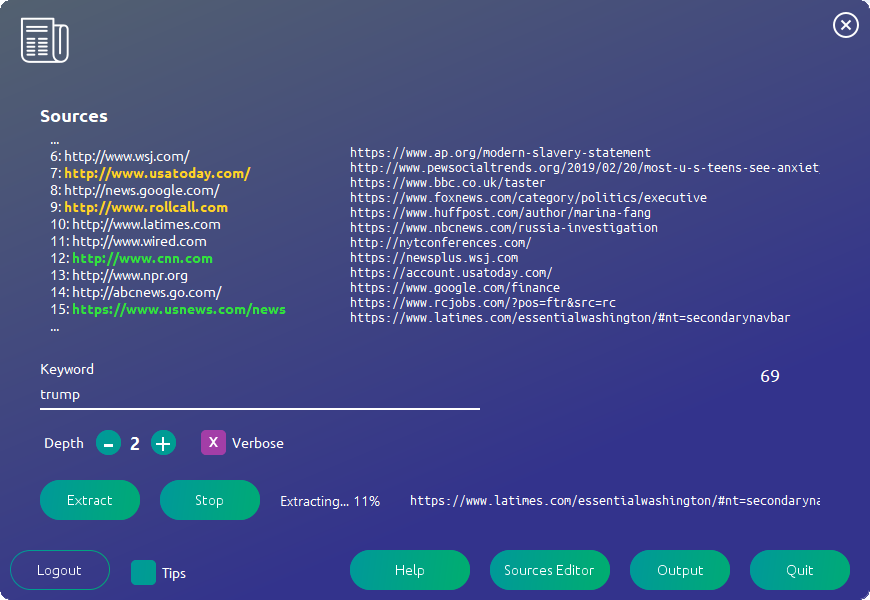
Designing a solution for a problem of this scope can be a very entertaining process. I decided to create it from the outside in – first starting with the GUI (Graphical User Interface), and then moving down into the “brain” of the program. I paid very close attention to detail and spent probably around 16 hours designing the GUI to look and feel very professional. After I had finished the bulk of the GUI, I transitioned to working on the search algorithm and interfacing that with each of the classes.

Object Design

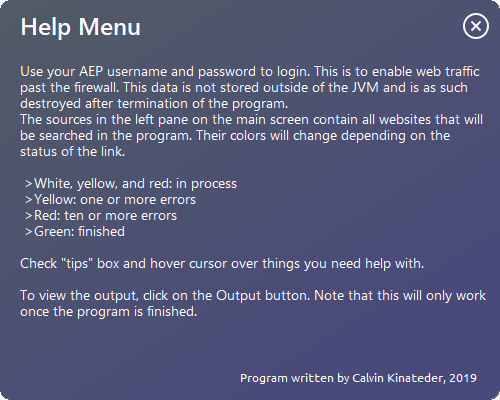
The program will have three forms: a login form (to use with the firewall), a main screen (to operate the extractors from), and a help menu. The color scheme for all three contains blue, grey, and green as the primary colors, and the font family is Ubuntu. The login screen is as follows:



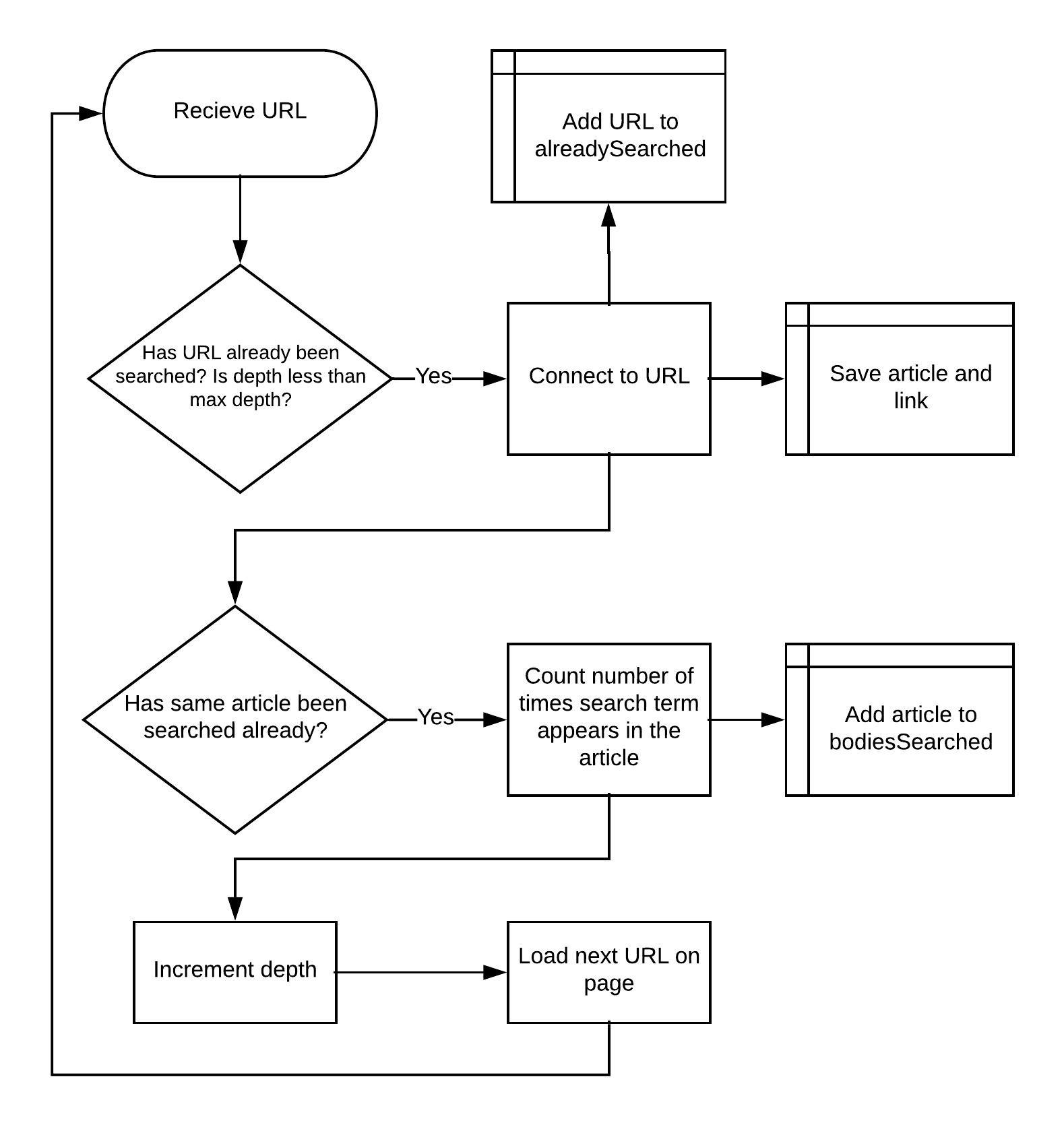
The interface for the main screen is designed for maximum information. The more the user knows about what is going on and how far the program has progressed, the better.

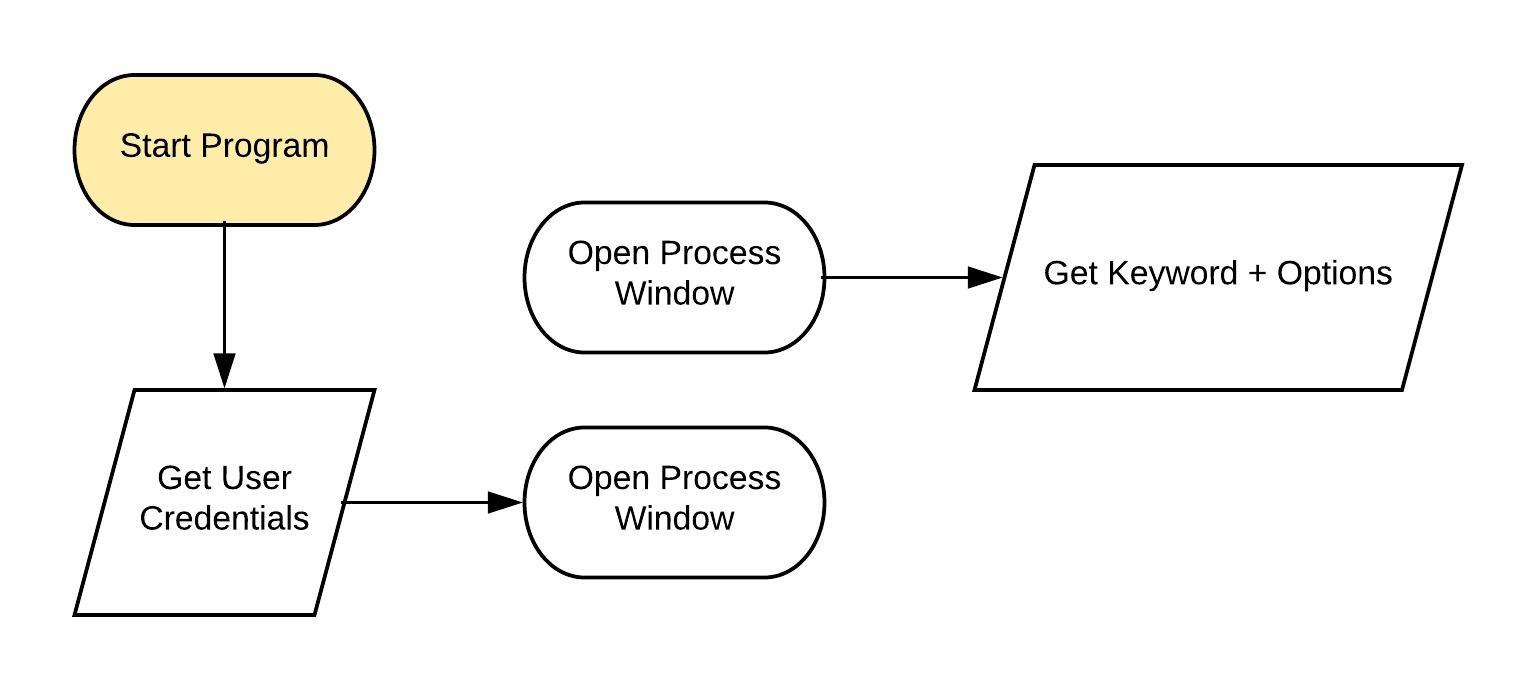
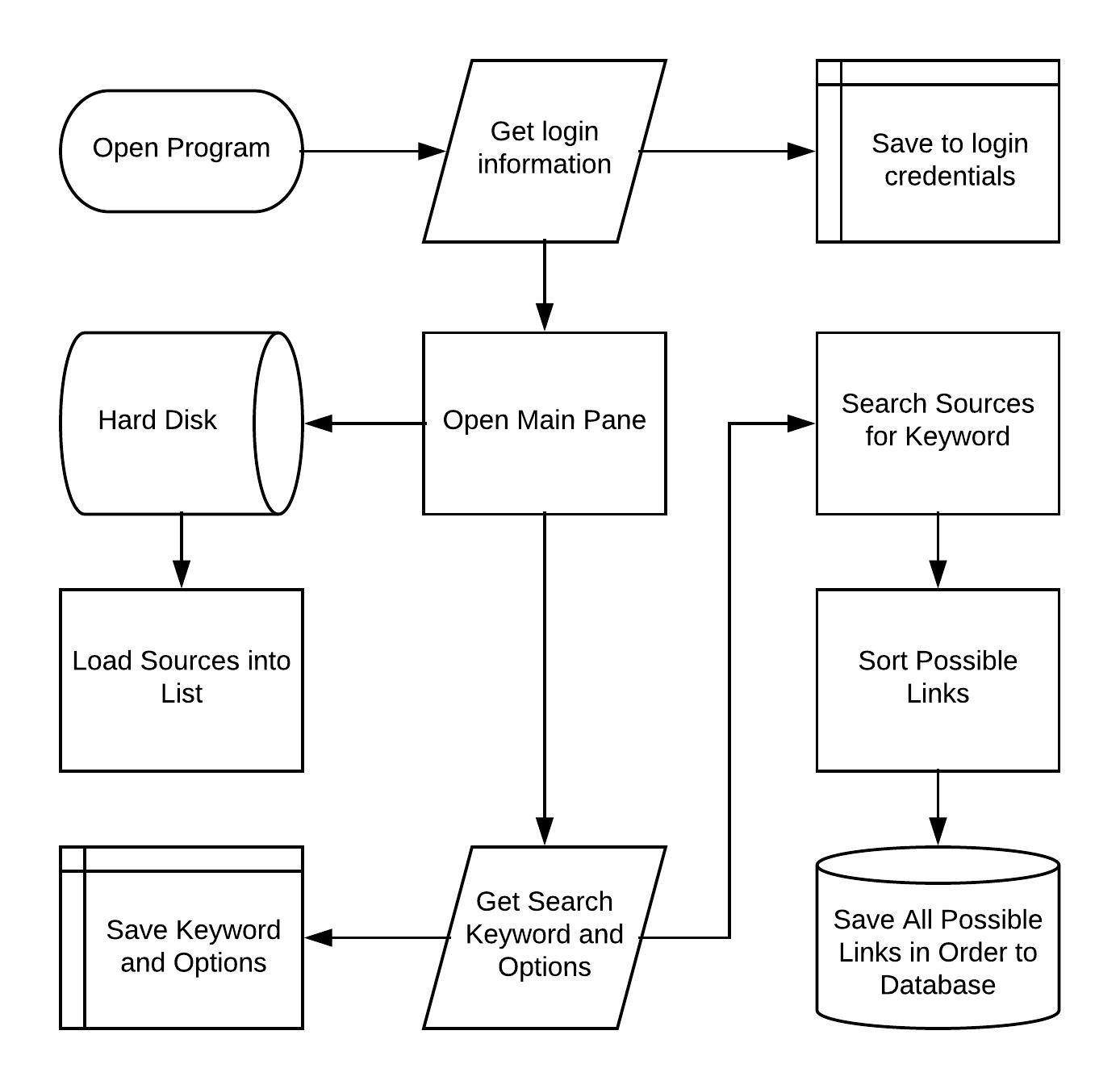


The help menu aims to provide clarity to any possibly vague instructions on the main program.

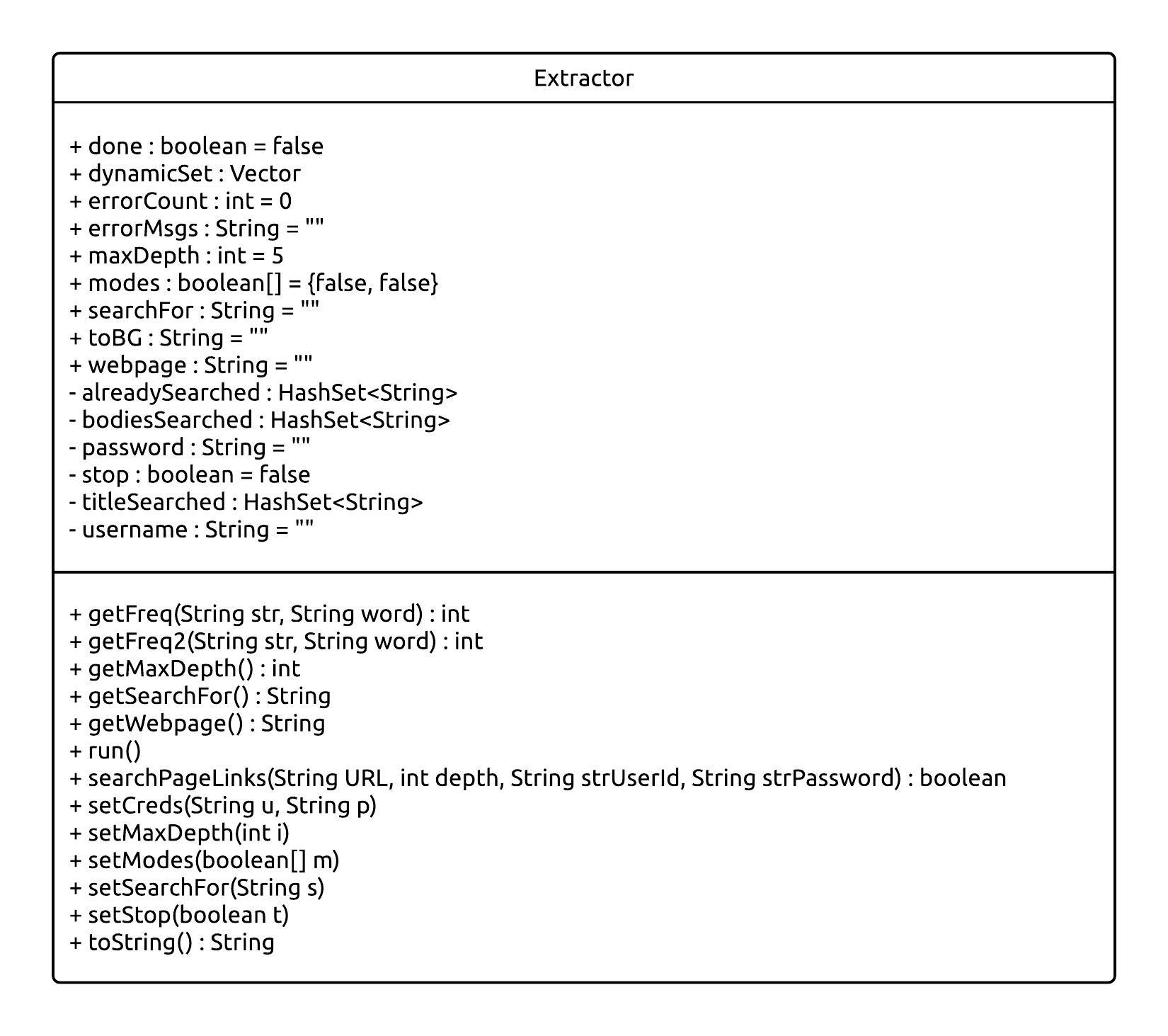


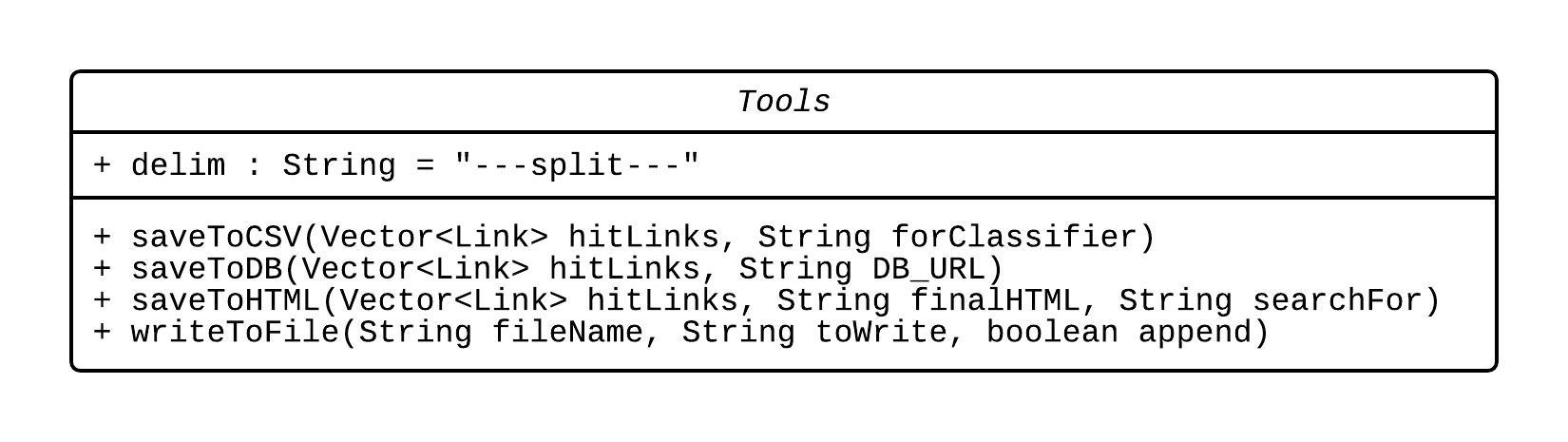
Flowcharts

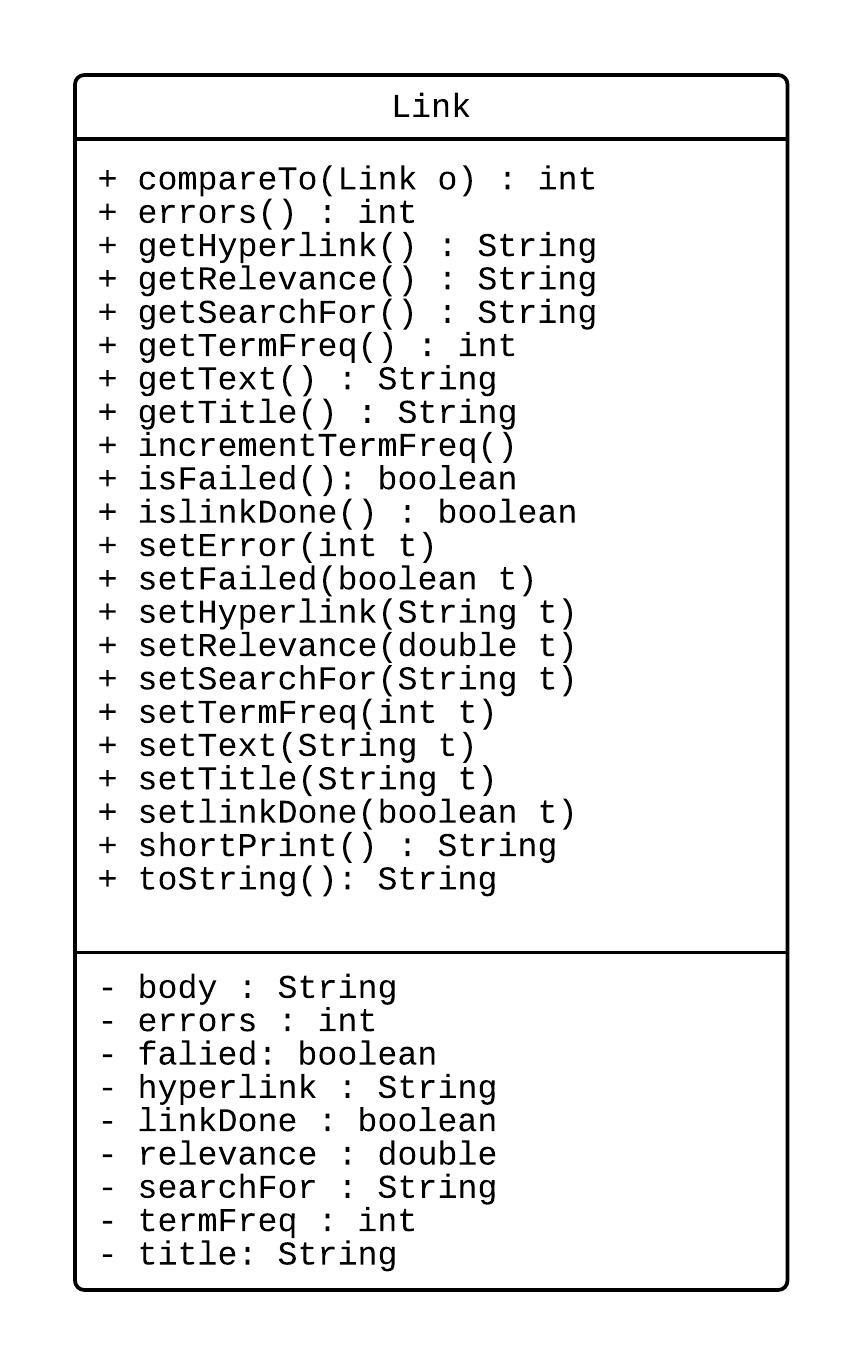
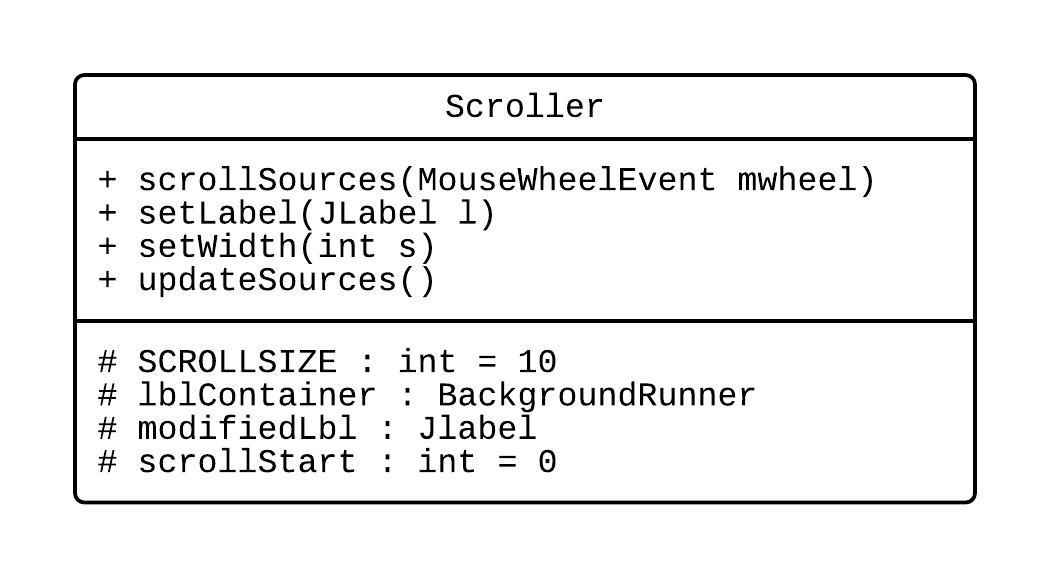
****



Class Diagrams

Using object-oriented programming, I will design this project with 9 classes. They will all interact with each other and make use of polymorphism, encapsulation, and inheritance. 





Class Overviews

There are 9 classes in StoryGrab, together totaling around 2700 lines of code.

**LoginWindow.java**

LoginWindow is the main class of the solution. It contains the code for the GUI as well as functions for the button actions. This class serves as the front-end and interface for the main algorithms.

**BackgroundRunner.java**

BackgroundRunner is the class that, using SwingWorker, allows the GUI to respond while the search algorithm is running. It is also responsible for controlling the multithreading and collecting all the data into a final list.

**Extractor.java**

The Extractor class contains the main searching algorithm. This method, which will be explained in more detail later, is a simple recursive bit of code.

**Scroller.java**

Scroller contains methods that work together to allow the user to control the lines that show up in a GUI panel with the mouse wheel. This is implemented in LoginWindow so that the user can see all the sources that are scanned and their corresponding statuses.

**AutomaticScrollUpdater.java**

To make the program more efficient, Scroller does not update every iteration of the main loop. Instead, it updates every 200 milliseconds. This class runs in the background and uses SwingWorker to be non-blocking.

**Tools.java**

Tools is the only abstract class in this project. It contains three functions: saveToCSV, saveToHTML, and saveToDB. They are used to save the output of the program to certain files to be viewed elsewhere.

**Link.java**

Link contains all the attributes of a Link object, such as title, hyperlink, and body. Implemented correctly, it is an easy way to collect all the data from one webpage and keep it all together.

**HelpMenu.java**

HelpMenu is a GUI form that gives the user help when they ask for it.

**User.java**

User collects the data of a user, just username and password, and stores it in an object.

Development Pseudocode

These give a general overview of what the most critical methods do.

***Login:***

*String username*

*String password*

*username and password =respective text fields on the login screen*

*switch to main screen*

***Start extraction (from LoginWindow):***

*if not running:*

*reset all variables*

*pass output labels, datasets to loader //loader is BackgroundRunner object*

*updateSources*

*start extraction in loader*

*endif*

***Start extraction (from BackgroundRunner):***

*while not done:*

*for every source:*

*if source isn’t searching yet:*

*start it searching*

*endif*

*publish output to screen*

*endfor*

*endwhile*

***Search webpage:***

*initialize authentication string to username+password*

*initialize error messages to a blank string*

*if link hasn’t already been searched and depth is less than max depth and no stop flags have been thrown*

*mark URL as already searched*

*connect to the URL and create a document*

*if search term appears more than once in the document*

*Count the number and add it to the list of matches*

*endif*

*for every link on page*

*re-run this method with depth incremented by one*

*endfor*

*endif*

Test Plan

These tests are here to ensure the solution meets the requirements

|  |  |
| --- | --- |
| Action to test | method of testing/expected result |
| Login screen is effective | Try to login to the firewall and access the internet from the program |
| Main screen is fluid and easy to use | Try to move the screen around and play with it to make things crash |
| All button clicks work | Test clicking each button |
| Extracting algorithm is accurate | Test searches with different depths and keywords and see if the output is reasonable |
| Scroll method works | Try scrolling in the source pane and make sure it works right |
| Help menu is helpful | Give the program to a new user and see if they can operate it without help |
| Source editor does not cause problems | Try to add a source from the source.txt file and make sure it does not fail to the backup |
| Output is readable and accurate | Run the program and make sure the output makes sense |
| Stop button saves correctly | Try terminating the program during execution and make sure the data is not corrupted in any way |
| Quit button does not crash but exits cleanly | Try terminating the program with the “quit” or “cancel” buttons |
| Colors in the scroll pane make sense | Run the program and watch the colors in the scroll pane and make sure nothing unexpected happens |

Criterion C: Development

Basic Overview of Operation

StoryGrab uses many classes working tightly together to achieve the fluidity that it does. LoginWindow acts a net for every input, taking user feedback and distributing it to where it needs to go. Recall that StoryGrab takes an almost infinite number of sources to search for. Also familiarize yourself to the concept of an Extractor object: a shell for a recursive search algorithm that contains all methods needed to effectively search a webpage. This class saves the hits to an ArrayList called hits. Note that Extractor extends the Thread class. Extractor objects are not meant to be operated directly from LoginWindow. Instead, they are run from BackgroundRunner, which acts as a container and runner for each instance of the Extractor class (one for each source). BackgroundRunner also implements the SwingWorker class. This allows BackgroundRunner to process in the background and to be non-blocking. Now, running each extractor sequentially would be a mistake. This would make execution take much longer than it really has the potential to do with multithreading.

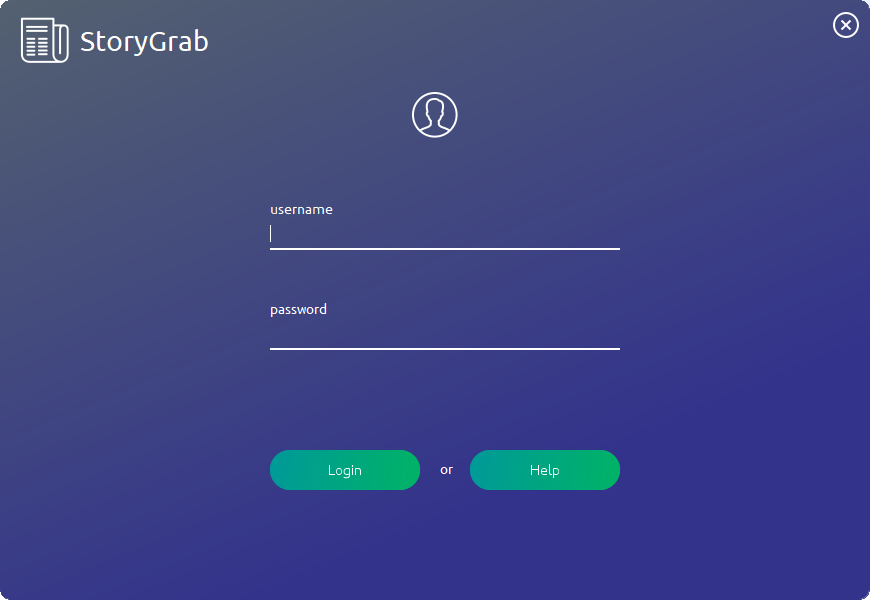
Because Extractor can be created as a Thread object, it retains Thread methods such as start() and join(). Extractor objects are all started individually on the search term defined in BackgroundRunner. Until they have all finished executing, the program (in BackgroundRunner) continuously checks if they are done. When they are all finished, they are joined together using Thread.join() and each respective hits is combined into a sorted final set called hitLinks. Using the static methods of abstract class Tools, this set is then saved to an HTML file to be viewed in a browser.

List of Techniques

This program uses all the following techniques:

* Graphical user interface
* Global and local variables
* Methods
* Recursion
* Search algorithms
* Merging data structures
* Polymorphism
* Inheritance
* Encapsulation
* Parsing a text file
* Use of additional libraries
* Inserting data into an ordered sequential file
* Abstract data type

Graphical User Interface

The GUI, as requested by the client, shows a visual representation of what is going on in the program.

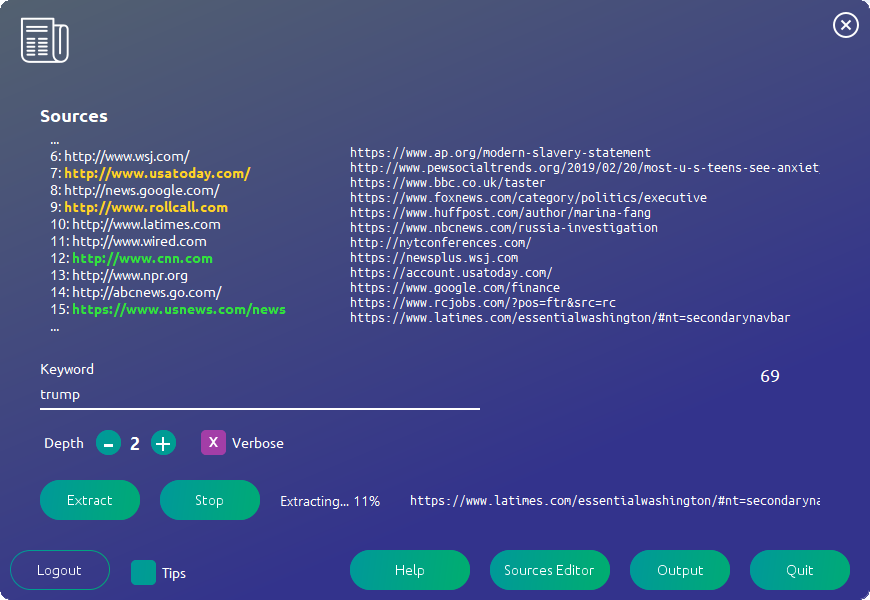
Close button

Help button

Login button

Password field

Username field



Progress label

Tips button

Logout button

Stop button

Extract button

Quit button

Output button

Source button

Help button

Depth controls

# of hits

Output line

Search field

Close button

Output pane

Sources pane

The first code piece is from the method doInBackground() in the class BackgroundRunner. It is placed here to illustrate the use of publish(List<String> chunks) to update the output panel. publish(List<String> chunks) sends chunks to process(List<String> chunks), which is reviewed in the second piece of code. This is because SwingWorker is a special class with reserved methods to be implemented. chunks is parsed and formatted to be more readable and then sent to the 3 output labels, referenced as longout, outputlbl, and hitslbl. hitslbl contains the number of search results currently found. outputlbl contains one line, the most recent message, and longout contains the 12 most recent messages.

**for**(int i = 0;i<extractors.size();i++){

Extractor t=extractors.get(i);

**if**(!t.isAlive()&&!used.contains(t)){

used.add(t);

leftToDo--;

*//set tmp vars*

double s = extractors.size();

double l = leftToDo;

double sm = s-l;

findSource(t.getWebpage()).setlinkDone(**true**);

System.out.println();

System.out.println("Done: ");

**for**(Extractor e : used){

System.out.println("\t"+e);

}

System.out.println();

percent=(int) ((sm/s)\*100);

statusLblRef.setText("Extracting... "+percent+"%"); *//set label*

}

publish(t.toBG);*//publish output to process*

**if**(!t.errorMsgs.equals("")){

publish("<font color=FFD126>"+t.errorMsgs+"</font>");

findSource(t.getWebpage()).setError(t.errorCount);

}

**if**(shouldStop){

publish("Cancelled by user - syncing threads");

**for**(Extractor ts : extractors){

publish("Extractor termination on "+ts.getWebpage());

ts.setStop(**true**);

}

**for**(Thread tt : extractors){

tt.join();

}

publish("Saving to database");

shouldStop = **false**; *//so loader can be used again*

isRunning = **false**; *//update this*

**return** **false**;

}

}

process(List<String> chunks) is a **protected** method that is meant to be modified as part of SwingWorker. Note that outputlbl, longout, and hitslbl are all of type JLabel and thus use the method setText().

**protected** void process(List<String> chunks) {

*// Get Info*

String currentOut = "";

currentOut = chunks.get(chunks.size()-1);

outputlbl.setText("<html>"+currentOut+"</html>");

int st = chunks.size()-OUTSIZE;

**if**(st<0){

st=0;

}

**if**(verbose){

String bo; *//no scroll*

bo = "<html><br>";

**for**(int i = st; i<chunks.size();i++){

String se = chunks.get(i).replaceAll("Searching ", "");

bo=bo+se+"<br>";

}

bo+="</html>";

longout.setText(bo);

}

**else**{

longout.setText("");

}

hitslbl.setText(*/\*"Hits: "+\*/*countAll()+"");

**if**(dynamic){Tools.saveToHTML(hitLinks, finalHTML, searchFor);}

}

Variables

Both public variables and private variables are used in most every class of the program, so I will not list every one out here. However, I will list ones that provide good examples for my complexities.

From the Extractor class:

**private** String username="";

**private** String password="";

**public** String toBG = "";*//to send to the backgroundrunner*

**public** String errorMsgs = "";*//to send to bg too*

**public** ArrayList<Link> hits = **new** ArrayList<>();

The public variables are always allowed to be modified from anywhere in the entire project, but the private variables are unable to be accessed directly from any other classes. This is called encapsulation. Using both global and local variables help to increase clarity throughout the program by making variables that don’t *need* to be accessed unable to be accessed. This way the developer is not confused in this instance.

Search Algorithm

The most import method in this entire project is the search algorithm. This method is the basis for the user to search the given source, URL, for the class variable searchFor. It is also a great example of recursion, abstract data types, and use of additional libraries.

**public** boolean searchPageLinks(String URL, int depth, String strUserId,

String strPassword) {*//returns true when done*

errorMsgs = "";

String authString = strUserId + ":" + strPassword;

String encodedString = *//need to authenticate for firewall*

Base64.getEncoder().encodeToString(authString.getBytes());

**if** ((!alreadySearched.contains(URL) && (depth < maxDepth))&&!stop) {

toBG=""+URL+"\n";*//add to toBG*

**try** {

alreadySearched.add(URL); *//add link to the hashset*

Document document = Jsoup.connect(URL) *//connect to doc*

.header("Authorization", "Basic " + encodedString)

.get();

Elements alreadySearchedOnPage = document.select("a[href]");

Elements txt = document.select("p");

String article = txt.text();

String title = document.title();

**if**(getFreq(article, searchFor)>1

&&!bodiesSearched.contains(article)

&&!titlesSearched.contains(title)){

int f = getFreq2(article, searchFor);

hits.add(**new** Link(URL, article,searchFor,title,f));

}

bodiesSearched.add(article);

titlesSearched.add(title);

depth++;

**for** (Element page : alreadySearchedOnPage) {

searchPageLinks(page.attr("abs:href"),

depth,strUserId,strPassword);

}

} **catch** (IOException | IllegalArgumentException e) {

System.err.println("For '" + URL + "': " + e.getMessage());

errorMsgs = "E:"+URL;

errorCount++;

}

}

**return** **true**; *//reached finish*

}

Recursion

Recall that recursion is “a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first.[[1]](#footnote-1)” searchPageLinks() meets this definition because it calls itself within the last for loop. The limit to the method not causing a stack overflow is that depth is decremented each time. The line

**if** ((!alreadySearched.contains(URL) && (depth < maxDepth))&&!stop) {...}

ensures that the method will not advance unless depth < maxDepth, saving the method from an overflow error.

Use of Additional Libraries

This program could not be possible without JSoup.[[2]](#footnote-2) JSoup is an external library that provides a framework to analyze webpages with. It contains functions that parse the webpage and can separate the title and other junk on the page from the body, which is immensely useful for StoryGrab.

Document document = Jsoup.connect(URL) *//connect to doc*

.header("Authorization", "Basic " + encodedString)

.get();

Elements alreadySearchedOnPage = document.select("a[href]");

Elements txt = document.select("p");

Document and Elements are data structures created by JSoup.

Advanced Data Types

Extractor uses HashSet to prevent duplicate links from being saved. HashSet is generally faster than other Collection-type data structures.[[3]](#footnote-3)

**private** HashSet<String> alreadySearched; *//no double searching*

Note that a default value is initialized within the class constructors. Use case:

**if** ((!alreadySearched.contains(URL) && (depth < maxDepth))&&!stop) {

...

alreadySearched.add(URL); *//add link to the hashset*

...

}

Parsing a Text File

StoryGrab’s sources are stored in a text file outside of Java. FileReader and BufferedReader allow for this data to be read from the Background class.

**public** void updateSrc(){

**if**(!isRunning){

sources.clear();

**try** {

*// FileReader reads text files in the default encoding.*

FileReader fileReader =

**new** FileReader(sourcesFile);

BufferedReader bufferedReader =

**new** BufferedReader(fileReader);

String line;

**while** ((line = bufferedReader.readLine()) != **null**) {

sources.add(**new** Link(line));

}

*// Always close files.*

bufferedReader.close();

}

**catch**(FileNotFoundException ex) {

System.out.println(

"Unable to open file '" +

sourcesFile + "'");

backupSrcs();

}

**catch**(IOException ex) {

System.out.println(

"Error reading file '"

+ sourcesFile + "'");

backupSrcs();

}

}

}

A new Link object for each line in the file is created and added to sources. Note that sources is

ArrayList<Link> sources = **new** ArrayList<>();

Multithreading

As previously stated, Storygrab relies on multithreading for its high speed and efficiency. This is done by creating an ArrayList of extractors and iterating through it whenever they all need to be acted on in the same way. This is all done in a non-blocking way.

**for**(Extractor e : extractors){

e.setModes(**new** boolean[] {**true**, **true**});

e.start();*//make new thread for each extractor*

}

However, the join method,

**for**(Thread tt : extractors){

tt.join();

}

is technically blocking – but as this is within a SwingWorker, it is non-blocking to the rest of the program.

Merging Data Structures

Merging multiple data structures in a program is actually quite simple given that they are all already in an ArrayList.

**public** void combine(){ *//only call when joined*

**for**(Extractor e : extractors){

hitLinks.addAll(e.hits);

}

}

This list is then sorted in cleanup().

Inserting Data into an Ordered Sequential File

This is not needed to be done until the end, when everything is processed and ready for output.

**public** String cleanup(){

combine();

shouldStop = **true**;

String out = "";

Collections.sort(hitLinks);

out+=Tools.saveToCSV(hitLinks, forClassifier)+"<br>";

out+=Tools.saveToHTML(hitLinks, finalHTML, searchFor)+"<br>";

out+=Tools.saveToDB(hitLinks, DB\_URL);

**return** out;

}

cleanup() is called as the program finishes up execution, to save the data that had been gathered.

**public** **static** String saveToCSV(List<Link> hitLinks, String forClassifier){

FileWriter fileWriter =

**null**;

**try** {

fileWriter = **new** FileWriter(forClassifier,**false**); *//add true to append*

BufferedWriter bufferedWriter =

**new** BufferedWriter(fileWriter);

**for**(Link l : hitLinks){

bufferedWriter.write(Tools.delim+l.getRelevance()+Tools.delim); *//delim -s ','*

bufferedWriter.write(l.getHyperlink()+Tools.delim);

bufferedWriter.write(l.getText()+"\n");

}

bufferedWriter.close();

fileWriter.close();

**return** "Successfully wrote to CSV";

}

**catch** (IOException ex) {

Logger.getLogger(BackgroundRunner.class.getName()).log(Level.SEVERE, **null**, ex);

**return** "Could not write to CSV";

}

}

Encapsulation

About half of the variables in the project are used privately. For these variables, the only way to access them is with get and set methods. The set methods are void and only set the variable, and the get methods return the variable type and set no variables. Both are essential in accessing their respective variables within the project.

Definitions:

**private** String searchFor;

**private** double relevance;

**private** boolean linkDone;

Accessors/Modifiers:

**public** String getSearchFor(){

**return** searchFor;

}

**public** double getRelevance(){

**return** relevance;

}

**public** boolean islinkDone(){

**return** linkDone;

}

**public** void setSearchFor(String t){

**this**.searchFor = t;

}

**public** void setRelevance(double t){

**this**.relevance = t;

}

**public** void setlinkDone(boolean t){

**this**.linkDone = t;

}

Inheritance

The Extractor class inherits (extends) Thread. Use of the Thread class allows for multithreading within the program.

**public** **class** **Extractor** **extends** Thread {

...

}

This way, Extractor can use the methods and properties of Thread in the project without re-writing that whole class.

Polymorphism

By the principle of polymorphism, inherited classes can change an inherited method from the superclass. This allows for subclasses to be further customized to better fit their needs. For example, the Extractor class extends Thread. Extractor gains properties of the superclass but is able to change some of the methods, such as run() and toString(). At runtime, the compiler will choose to use the most recently defined definition of the extended methods rather than the base class definitions.

run() runs the whole class:

**public** void run() {

Thread.currentThread().setName(webpage);

System.out.println("Extractor running on "+webpage+" at max depth "

+maxDepth);

done = searchPageLinks(webpage, 0, username, password);

**if**(stop){

System.out.println("Extractor prematurely finished on "+webpage);

}

**else**{

System.out.println("Extractor client successful on "+webpage);

}

}

toString() defines what will be returned when the initialized object is printed:

**public** String toString(){

**return** "Page: "+webpage+", Depth: "+maxDepth;

}

1. <https://www.merriam-webster.com/dictionary/recursion> [↑](#footnote-ref-1)
2. <https://jsoup.org/> [↑](#footnote-ref-2)
3. <https://www.javacodegeeks.com/2010/08/java-best-practices-vector-arraylist.html> [↑](#footnote-ref-3)