Axle Addict

Edgar Oregel

Sukhdip Singh

Drew Overgaard

Peter Rogers

Jaskaran Virk

Navjot Singh

(2) Team Member & Contribution

- Edgar Oregel

- UI and frontend developer
- Personality Window: Made window, displayed personality test, UI design, functionality for question input
- Main Window: Made window, UI design, functionality
- Exit alert for all windows
- Personality Results Window: Made window, UI design
- Formulate Calculator equation with research

- Navjot Singh

- Put all of the backends together for the personality feature, so that keywords were made into sentences and google api was called using those sentences. The results that the google api returned were then parsed for makes and models and then further refined to get the top 5.
- Made all the Make, Model, and year parsers in the Car API class, which in turn helped the car search make searches more efficiently.
- Set up the backend for the car calculator
- Set up the user interface for the car calculator
- Plugged the backend and frontend for the car calculator.
- Tested, the api classes and car calculator backend.

- Sukhdip Singh

- Create Personality Window
- Add over 100 keywords to match a user's personality
- Create database to store keywords
- Create Personality multiple choice and written text questions to ask user's and be able to pull keywords from those questions

- Create JUnit test cases for database
- Create color scheme for all windows

- Jaskaran Virk

- Using Algorithms, parsed the API data to get certain keywords from the database to complete the Personality test
- Implemented the Boyer-Moore algorithm with the project, but ended up using Hashmaps and arraylists of strings
- Made API objects to get the # of occurrences of each key word for Personality Results
- Basic UI Color Schemes
- Dynamic Sizing of buttons on the three main classes (along with Peter/Edgar/Navjot)

- Peter Rogers

- Implementing keyword matching; wrote tests to confirm functionality
- Integrated keyword matching with database
- Basic implementation of personality results window
- Assisted with implementing SQLite database and functions for accessing
- Research and attempted implementation of dynamic resizing (discarded at group agreement)

- Drew Overgaard

- Researching and implementing google API; early stages
- Worked on the Car Search window functionality and design
- Worked with the Edmunds API to grab car specifications and images
- Worked on displaying results in the personality window

(3) Project Objectives

- Value delivered to your client

- Connect a user’s personality to a set of cars based on user entries to a personality test
- Calculate a car’s 0-60 and quarter mile time

- User can search for car’s based on make, model, year and get relevant information about the searched car
- Overall an easy application where anyone can find information about cars they’re interested in and find a car that might fit their personality if they’re indecisive

(4) Project Design

- Team goals

- Build software that can be easily understood and modified later by someone not involved in the project.
- Build code that can be reused outside of class rather than it being a class project and nothing more.
- Organize our work in a way that anyone can look at our work board and be able to identify exactly what we’ve done, are currently doing, and what is left to do.
- Communicate with our team to be able to understand the tasks and work that every team member is doing individually
- To have fun and enjoy implementing something cool

- User stories completed

-Edgar Oregel

- Add units to car search results
- Fix the exit alert pop-ups
- Fix main window elements
- Algorithm modification
- Increase the size of the text on personality test
- Dynamic sizing to static sizing: calculator
- Dynamic sizing to static sizing: personality results window
- Have people take survey for personality questions

- Create new window for personality test results
- Get keywords from personality test input
- Front-end testing
- Car search UI test
- Personality UI test
- Get the keywords matching working
- Fix personality backend
- Change input to keywords matching to user personality test input
- Save personality test results
- Work on finalizing UI for personality window
- Randomize the questions added into window
- Finish up UI for main window

-Navjot Singh

- Parse personality results cars
- Fix Personality random bad sizing
- Make every title the same
- Spell Check
- Change the loading in personality and personality results
- Prioritizing all cars (by occurrence in the google API)
- Formulate a better search string for the google API call
- Make google API call for personality test
- Constructing sentence to search google API
- Save Edmund's API call to file
- API parsing testing
- Back end testing Epic

- Revise Google API Parsing
- Parse Edmund's API data
- Create Edmund API call to pull the rest of Vehicle info
- Car calculator backend testing
- API Tests
- Integrate front end with back-end in the calculator
- Car Search API information
- Create UI for car time calculator
- Create mock up background for main window
- Calculate users car’s 0-60/quarter mile
- Json Parser for API’s
- Pull from car API

-Sukhdip Singh

- Null check for personality DB
- Readjust window components(Main Window)
- Fix main page size of elements and window
- Dynamic sizing to static sizing: Main Window
- Dynamic sizing to static sizing: Car Search
- Dynamic sizing to static sizing: Personality Window
- Standardize UI Color Scheme (Make UI Look Better)
- Expand database for modifying keywords
- Writing data from API’s into files
- Personality question validation
- Algorithm for gettings cars from users personality
- Database testing

- Reword the questions to have the user enter better searchable keywords
- Database insert keywords
- Read keywords from file for DB
- Research Synonym Finder API
- Clean SQL functions
- Create mock up background for main window
- Research and test different personality algorithms

Jaskaran Virk

- Add Scrollview to question window
- Research Car Search API information
- Display Scrollable pane for Personality Class
- Create Search String algorithm for Google API string parsing
- Implement Boyer-Moore Algorithm for Simple search strings
- Use this algorithm to search for certain keywords from the database
- Have people take survey for personality questions
- Print Google Car results to the Personality Results Window
- Algorithm Modification (Hashmaps and Arraylists)
- Change the color pattern of the Personality results
- Modify back Buttons to have it in the same location for everyone

Peter Rogers

- Fix main window sizing error
- Dynamic UI sizing and positioning
- Make a JUnit test class for keyword matching
- Implement database with keyword matching function
- Get familiar with JUnit

- Use sample strings and keywords to match to database testing
- Research and test different personality algorithms

Drew Overgaard

- Personality results car details
- Make car search image larger
- Resize the car search buttons/text
- Refactor car search backend
- Add in a default car image
- Include car trim pictures
- Returning the cars to the user
- Make the car search window results dynamic
- Implement Edmund's API into search
- Display the car to the user for car search
- Parse Edmund's API data
- Update car search UI
- Create Edmund API call to pull the rest of the Vehicle info
- Return JSON results from Google API
- Create UI for window matching users car specs with cars

(5) Implementation Details

- Programming Languages Used

- Java

- Libraries or Tools Used

- SQLite
- Edmunds API
- Google Custom Search API

- JUnit & Hamcrest
- JSon
- JavaFX

- Challenges and Solutions

- One of our biggest challenges was overcoming Edmunds api call limit. We had to be very careful with our calls and only make them when we had to. We made a file that had information on make, model, and year of all cars (from edmunds api).
- The personality feature was a very big challenge for the team. We did a lot of research on how to match personality with cars and how it would look on our application. We looked at things like natural word processing, and in the end decided that we would use key words. We ended up using keywords because the team thought that was something realistic and achievable and we were able to achieve it.
- Making clear and concise stories was also a challenge for us, in the beginning. A lot of our stories were to general and often times didn't make sense. Team members would understand the story when it was planning time but let when it came time to implement they would forget and because the story descriptions were not very clear they would be stuck until they received further clarification. Afterwards we all started spending more time on each story making sure it was as clear as possible, this helped a lot because team members were able to use the description of their stories to figure out what they were doing for that sprint.
- Staying on top of our work and making constant progress was a challenge in the beginning of our project. We were all assigned our individual stories/tasks and expected to finish everything within a certain time. In order to make sure that a lot of progress was made early into the sprint, we started our weekly meetings where we all met and devoted all of our time to working on the project to make significant progress. This was held every Thursday of every week the second half of the project.

- Effective communication within the team on slack was also a very big issue. Some team members were at times confused as to who was doing what as well as knowing where we were on our sprint progress. Some team members at times had other circumstances arise that they weren't able to finish their tasks/stories. Other times, some had troubles finishing it because they were lost and confused, however due to the lack of effective communication, it was never voiced until the last day of the sprint during class which defeated the purpose of every team member knowing the status of the rest of the team and being able to help others to finish the sprint tasks. We resolved this by mandating every team member to post an update on slack at least every other day. This helped the team know the status of everyone and be able to help others who needed.

(6) Testing

- For all of our tests we did white box testing.
- We wrote classes first then tested them for different use cases.
- API's were tested to make sure they were working (returning appropriate http status code)
- Parser's were given text files so we new the expected outcome so we could test that
- For the other classes we tested core functions that required had a lot of conditional statements, and we tried to hit most of the use cases.

- Test Cases

Proof of testing:

Story #/Name: Google API Test
Tester's Name: Edgar

Description of Acceptance Criteria to be followed or Acceptance Test to Perform:
Google API test checking that the api calls are successful so that calls can return information to return to the user.

Assumptions one needs to make: None

Variations	Expected Results	Comments	Done?

Make Google API call and return a 201 status code	Google custom search api returns a 201 http status code	We send a basic request to the Google custom API and check for a 201 http status code	Yes

```
public class GoogleAPITest {
	private GoogleAPI google;
	@Test
	public void test() {
		// one simple test to make sure the google api is working
		google = new GoogleAPI("Testing connection");
		assertEquals(200, google.getHttpCode());
	}
}
```

Story #/Name: Database testing
Tester's Name: Sukh

Description of Acceptance Criteria to be followed or Acceptance Test to Perform:

Adding Items to DB

1. Add Item to a table
2. DB gets updated
3. a confirmation string is shown or an error message is shown

Removing Items to DB

1. remove an Item from a table
2. DB gets updated
3. a confirmation string is shown or error message is shown

Updating Items to DB

1. select an value to update from a table
2. DB item updates that value
3. a confirmation string is shown or error message is shown

Searching Items In DB

1. Select a value to search from table
2. DB searches table
3. result is shown or an empty string gets returned

Assumptions one needs to make:

No assumptions

Variations	Expected Results	Comments	Done?
Try connecting database	Database gets successfully connected a string is returned		Yes
Database table gets dropped	Empty string is returned when selecting data		Yes

// Designed to test inserting data into the database

```
assertTrue(SqlKeywords.insertData("keywords", "Tall", "Random") == true);
```

// Designed to test opening the file to read in keywords

```
assertTrue(SqlKeywords.addKeywords("Resources/textfile.txt") == false);
```

Story #/Name: #35 API Tests
Tester's Name: Navjot Singh

Description of Acceptance Criteria to be followed or Acceptance Test to Perform:

Parse a text file containing data from a google API call and return. Parse a file containing empty brackets.

Assumptions one needs to make:

Assume that the call to Google API completed successfully, meaning, there was no http error from an invalid call.

Variations	Expected Results	Comments	Done?
Test the Google API parsing function by parsing a text file containing	File will be parsed successfully, returning the title	Data to be parsed saved in a text file, so that an API call	Yes

a call in JSON format.	and snippet from the JSON file.	does not have to be used when testing.	
Parse file containing only empty brackets {}	Should not return anything, since the parsed file does not contain the keywords ‘title’ or ‘snippet’.	None	Yes

```
@Test
public void test1() throws JSONException {

		String str = "";
		try {
			str = readFileToString("testData/GoogleAPITestingDocs/Test1.txt");
		} catch (IOException e) {
			e.printStackTrace();
		}
		assertNotEquals("", str);
		JSONObject test1 = new JSONObject(str);

		googleApi = new GoogleAPI("family car");
		String parsedOutput = googleApi.googleAPIParser(test1);

		try {
			str = readFileToString("testData/GoogleAPITestingDocs/Test1Results.txt");
		} catch (IOException e) {
			e.printStackTrace();
		}
		assertNotEquals("", str);
		assertEquals(str.length(), parsedOutput.length());
}

@Test
public void test2() throws JSONException {

		String str = "";
		try {
			str = readFileToString("testData/GoogleAPITestingDocs/Test2.txt");
		} catch (IOException e) {
			e.printStackTrace();
		}
		assertNotEquals("", str);
		JSONObject test2 = new JSONObject(str);
```

```
                    googleApi = new GoogleAPI("family car");
                    String parsedOutput = googleApi.googleAPIParser(test2);

                    assertEquals(parsedOutput, "");
          }
}
```

Story #/Name: #Algorithm Testing

Tester's Name: Jaskaran Virk

Description of Acceptance Criteria to be followed or Acceptance Test to Perform:

Find multiple algorithms to find the best algorithm to parse through the API searches (along with Drew's code). Somehow integrate the Boyer-Moore algorithms to either parse API into a txt file or run algorithm separately to find certain keywords.

Assumptions one needs to make:

All information will be stored in a txt file, and the keyword(s) from the database will come match the personality class responses (still in progress), and then check the Google/Edmund's API to get results of matching words.

Variations	Expected Results	Comments	Done?
Get the certain index of the keyword needed, store it in an Arraylist	Keyword Index is given back to the user.	Searches for a single iteration of the word. In our case, we only need one to match it with the Google API result	Yes
Integrate Algorithm to store API info into txt file and get personality resulting keywords to compare the two	Check for same words and return words in an Arraylist to get successful keywords we want	In progress on how to connect the two classes and results	No

Boyer Moore Test: Test case is to find a single word in an String array

```
public Algorithms(String pattern){

        this.maxIndex = 256; //# of indexes
        this.pattern = pattern;

        textInput = new int[maxIndex]; //create new array with STRING indexes

        for (int k = 0; k < maxIndex; k++)
             textInput[k] = -1;
        for (int j = 0; j < pattern.length(); j++) //For the length of the pattern..
             textInput[pattern.charAt(j)] = j; //find the location of the specified string
    }


public int search(String text){ //search function was found online, with minor changes

        int string_length = text.length();
        int string_pattern = pattern.length();
        int skip;

        for (int i = 0; i <= string_length - string_pattern; i += skip){

            skip = 0;
            for (int j = string_pattern - 1; j >= 0; j--){

                if (pattern.charAt(j) != text.charAt(i + j)){

                    skip = Math.max(1, j - textInput[text.charAt(i + j)]);
                    break;
                }
            }

          if (skip == 0)
          return i;
        }

      return string_length;
    }


private void NoPatternMatch(String pattern){
                          int[] error_check = {-1};
                          int n = pattern.length(); //

                     for (int j = 1; j < n; j++)
                {
                    int i = error_check[j - 1]; //for the length of the pattern string - 1
                    while ((pattern.charAt(j) != pattern.charAt(i + 1)) && i >= 0) //If the subsequent string letter
doesn't match..
                        i = error_check[i]; //set that value into error_check array
                    if (pattern.charAt(j) == pattern.charAt(i + 1))
                        error_check[j] = i + 1; //the new array index holds the start of error string, or incorrect
pattern
                    else
                        error_check[j] = -1;
                }
            }
```

-Navjot Singh

Story #/Name: Car Calculator backend testing
Tester's Name: Navjot

Description of Acceptance Criteria to be followed or Acceptance Test to Perform:
The backend for the car calculator, it calculates the 0-60, and quarter mile from user's oil change frequency, year, hp, torque, weight, odometer, and stock check box.

Assumptions one needs to make: none

Variation	Expected Results	Comments	Done?
Users car is not stock and he/she does regular oil changes	Real hp to be the same as what user gives to program	Oil changes, car year, and car being stock affect the "actual hp" of the car	Yes
User has an older car and doesn't always change oil when supposed to	Real hp is less than what user gives the program	As the car gets older oil changes become more important, and if you keep missing them, it kills your horsepower	Yes
Users with new cars that don't change their oil regularly	Real hp is about the same as what the user provides	A car's horsepower doesn't deteriorate to fast when it's new	Yes

```
private CarCalcBackend backend = new CarCalcBackend();
```

```
	@Test
	public void testRealHP() {
		int hp = 300;
		int curYear = Calendar.getInstance().get(Calendar.YEAR);
		int tenYrs = curYear - 10;
		int fiftyYrs = curYear - 50;
		// new car with less than 50K miles and bought in the current year
		assertTrue(backend.realHP(hp, 0, 0, 0, curYear) == hp);
		// stock car
		assertTrue(backend.realHP(hp, 0, 0, 1, curYear) == hp-25);
		// really badly maintained car but still same year, should not affect
performance to much
		// therefore horsepower should still be relatively the same
		assertTrue(backend.realHP(hp, 5, 5, 0, curYear) == hp);
		assertTrue(backend.realHP(hp, 2, 2, 0, curYear) == hp);
		// a car that was badly maintained for 10 years will have more
horsepower than a car that was
		// badly maintained for 50 years
		assertTrue(backend.realHP(hp, 3, 3, 0, tenYrs) > (backend.realHP(hp, 3,
3, 0, fiftyYrs)));
		// a car that was very badly maintained for 10 years will have more
horsepower than a car that was
		// slightly badly maintained for 50 years
		assertTrue(backend.realHP(hp, 4, 4, 0, tenYrs) > (backend.realHP(hp, 1,
1, 0, fiftyYrs)));
	}
```

Variation	Expected Results	Comments	Done?
Users with cars that have higher hp and torque and less weight	Low quarter mile and 0-60 times	The more the power and less weight = faster car	Yes
Users with cars with a lot of weight	Higher 0-60 times. Quarter mile only slightly higher.	Cars that are heavier usually take longer to pick up speed	Yes
Stock vs a non-stock car	Stock will always lose	A car with aftermarket parts are almost always faster than its stock counterpart	Yes
Users enters in part	Fields that have no	N/A	Yes

of the fields	value show errors		
User enters invalid values for fields	Fields with invalid values light up and notify the user	N/A	Yes
User enters in correct values for fields and clicks calculate button	The backend gets the value and computes the 0-60 and quarter mile taking each	N/A	Yes

```
       @Test
       public void testQuarterMile() {
              int hp = 300;
              int weight = 3500;
              int curYear = Calendar.getInstance().get(Calendar.YEAR);
              int tenYrs = curYear - 10;
              int fiftyYrs = curYear - 50;
              // making sure weight and horsepower change the quarter mile of the car
              assertTrue(backend.quarterMile(hp, weight, 0, 0, 0, curYear) <
backend.quarterMile(hp, weight + 1, 0, 0, 0, curYear));
              assertTrue(backend.quarterMile(hp, weight, 0, 0, 0, curYear) <
backend.quarterMile(hp - 1, weight, 0, 0, 0, curYear));
              // car that is extremely badly maintained will still out put the same
amount of power because decrease in a cars power
              // happens overtime no matter how badly it is maintained.
              assertTrue(backend.quarterMile(hp, weight, 5, 5, 0, curYear) ==
backend.quarterMile(hp, weight, 0, 0, 0, curYear));
              // stock car will lose to the same car that has been tuned up (even by a
little)
              assertTrue(backend.quarterMile(hp, weight, 0, 0, 0, curYear) <
backend.quarterMile(hp, weight, 0, 0, 1, curYear));
              // a car that is older should output the same amount of power if it has
been maintained well
              assertTrue(backend.quarterMile(hp, weight, 0, 0, 0, tenYrs) ==
backend.quarterMile(hp, weight, 0, 0, 0, curYear));
              assertTrue(backend.quarterMile(hp, weight, 0, 0, 0, tenYrs) ==
backend.quarterMile(hp, weight, 0, 0, 0, fiftyYrs));
       }

       @Test
       public void testZeroToSixty() {
              int hp = 300;
              int weight = 3500;
              int curYear = Calendar.getInstance().get(Calendar.YEAR);
              int tenYrs = curYear - 10;
              int fiftyYrs = curYear - 50;
              // making sure weight and horsepower change the 0-60 of the car
              assertTrue(backend.zeroToSixty(hp, weight, 0, 0, 0, curYear) <
backend.zeroToSixty(hp, weight + 1, 0, 0, 0, curYear));
              assertTrue(backend.zeroToSixty(hp, weight, 0, 0, 0, curYear) <
backend.zeroToSixty(hp - 1, weight, 0, 0, 0, curYear));
              // car that is extremely badly maintained will still out put the same
amount of power because decrease in a cars power
              // happens overtime no matter how badly it is maintained.
              assertTrue(backend.zeroToSixty(hp, weight, 5, 5, 0, curYear) ==
backend.zeroToSixty(hp, weight, 0, 0, 0, curYear));
              // stock car will lose to the same car that has been tuned up (even by a
little)
              assertTrue(backend.zeroToSixty(hp, weight, 0, 0, 0, curYear) <
backend.zeroToSixty(hp, weight, 0, 0, 1, curYear));
              // a car that is older should output the same amount of power if it has
been maintained well
              assertTrue(backend.zeroToSixty(hp, weight, 0, 0, 0, tenYrs) ==
backend.zeroToSixty(hp, weight, 0, 0, 0, curYear));
              assertTrue(backend.zeroToSixty(hp, weight, 0, 0, 0, tenYrs) ==
backend.zeroToSixty(hp, weight, 0, 0, 0, fiftyYrs));
       }

}
```

Story #/Name: CarAPI Testing

Tester's Name: Navjot

Description of Acceptance Criteria to be followed or Acceptance Test to Perform:
Testing the http status code of the car API call.

Assumptions one needs to make:

Variation	Expected Results	Comments	Done?
Making a basic request to Edmunds API	Server returns a 200 http status code	To check if server is responding	Yes
Making a bad request to Edmunds api	Server will not return a 200 http status code	The test will fail	yes

```
@Test
       public void testCarAPIs() {
        // both arraylists initialized as empty lists
        ArrayList<String> emptyModels = testCarAPI.getCarModels();
        ArrayList<String> emptyTrims = testCarAPI.getCarTrims();
        // makes sure the NHTSA API works (function pulls from there API)
        assertTrue(testCarAPI.getCarMakes().size() > 0);
        assertEquals(200, testCarAPI.getHttpCode());
        // makes sure the NHTSA API works (function pulls from there API)
        assertTrue(emptyModels.size() == testCarAPI.getCarModels().size());
        // makes sure the Edmund's API works (function pulls from there API)
        assertTrue(emptyTrims.size() == testCarAPI.getCarTrims().size());
        testCarAPI.setCarModels("HONDA");
        testCarAPI.setCarTrims("HONDA", "CIVIC", 2010);
        // makes sure the getters and setters of the class work correctly
        assertFalse(emptyModels.size() == testCarAPI.getCarModels().size());
        assertFalse(emptyTrims.size() == testCarAPI.getCarTrims().size());
    }
```

Story #/Name: CarAPI Testing

Tester's Name: Navjot Singh

Description of Acceptance Criteria to be followed or Acceptance Test to Perform:
Parsing the carAPI calls, to get Make, model, and year.

Assumptions one needs to make: Makes, Models, and Year for all cars in the Edmunds API ir copied into a text file to reduce calls to the API.

Variation	Expected Results	Comments	Done?
Trying to get all the makes from the API, with a full response of data.	Get an ArrayList of all makes from the Edmunds API	Server returns a huge json with make, model, and year data. The goal of this test is to test the parser that parses the data	Yes
Getting models for an existing make	Get an ArrayList of models for the specified model	The function has the data from the get make call and parses that data to get the model, in order to avoid	Yes

		making unnecessary calls.	
Getting all years for a make and model	Get an ArrayList of models for the specified model	Uses model ArrayList to find the index that will be used to find the model in all of the data, then parses that data to find years	Yes
Getting an empty json	Returns empty ArrayList	To make sure the parser could handle no data	Yes

```
       @Test
       public void testCarAPIParsers1() throws JSONException, IOException{
              String str = "";
              try {
                     str = readFileToString("testData/CarAPITestingDocs/Test1.txt");
              } catch (IOException e) {
                     e.printStackTrace();
              }
              assertNotEquals(str,"");
              JSONObject json = new JSONObject(str);
              ArrayList<String> makes = testCarAPI.carAPIParserStr(json, "makes",
"niceName");
              assertNotNull(makes);
              assertNotEquals(makes.size(),0);


              try {
                     str = readFileToString("testData/CarAPITestingDocs/Test2.txt");
              } catch (IOException e) {
                     e.printStackTrace();
              }

              assertNotEquals(str,"");
              json = new JSONObject(str);
              ArrayList<String> models = testCarAPI.carAPIParserStr(json, "models",
"niceName");
              assertNotNull(models);
              assertNotEquals(models.size(),0);

              try {
                     str = readFileToString("testData/CarAPITestingDocs/Test3.txt");
              } catch (IOException e) {
```

```
                        e.printStackTrace();
                }

                assertNotEquals(str,"");
                json = new JSONObject(str);
                ArrayList<Integer> years = testCarAPI.carAPIParserInt(json, "years",
"year");
                ArrayList<Integer> results = new ArrayList<Integer>();
                for (int i =0; i< 19; i++){
                        int a = 1997;
                        results.add(a+i);
                        int actualYr = years.get(i);
                        assertEquals(a+i, actualYr);
                }
                assertNotNull(years);
                assertNotEquals(years.size(),0);
                assertEquals(results.size(),years.size());
        }

        @Test
        public void carAPIParserTest2() throws JSONException{
                String str = "";
                ArrayList<String> empty = new ArrayList<String>();
                try {
                        str = readFileToString("testData/CarAPITestingDocs/Test4.txt");
                } catch (IOException e) {
                        e.printStackTrace();
                }
                assertNotEquals(str,"");
                JSONObject json = new JSONObject(str);
                ArrayList<String> makes = testCarAPI.carAPIParserStr(json, "makes",
"niceName");
                assertEquals(makes,empty);
        }
}
```

-Sukhdip Singh

1. assertTrue(SqlKeywords.connect() == true);
2. assertTrue(SqlKeywords.createNewDatabase() == true);
3. assertFalse(SqlKeywords.createNewTable("table") == true);
4. assertTrue(SqlKeywords.selectData("keywords") == true);
5. assertFalse(SqlKeywords.selectData("tempTable") == true);
6. assertTrue(SqlKeywords.selectData("newtable") == false);
7. assertTrue(SqlKeywords.insertData("keywords", "Tall", "Random") == true);
8. assertFalse(SqlKeywords.insertData("tempTable", "Tall", "Random") == true);
9. assertTrue(SqlKeywords.insertData("temp", "name", "cat") == false);
10. assertTrue(SqlKeywords.removeData("keywords", "category", "Sport") == true);

11. assertFalse(SqlKeywords.removeData("tempTable", "category", "Sport") == true);
12. assertTrue(SqlKeywords.removeData("temp", "cat", "name") == false);
13. assertTrue(SqlKeywords.addKeywords("Resources/Keywords.txt", "..Resources/Keywords.txt") == true);
14. assertFalse(SqlKeywords.addKeywords("Resources/temp.txt", "../Resources/temp.txt") == true);
15. assertTrue(SqlKeywords.addKeywords("Resources/textfile.txt", "../Resources/textfile.txt") == false);

(7) Project Highlights (Retrospective)

- Parts of the actual software you are proud of

- We are proud that our application allows users to search for cars and view the most relevant information about those cars.
- That our software has multiple features to it
- We are proud that our application matches users to cars based on the information they enter in the personality tests that we designed.
- Learning how to use the API's and being able to successfully pull any and all information that we needed about cars and even getting the picture of the car we return.
- Learning new things in Java and being able to apply them to our project, (like JavaFX, JSon, Junit, etc.)

- Things you guys did as a team that you think worked really well

- As a team we decided to meet up every Thursday to work on the project. Meeting in person allowed us to more easily communicate. It also allowed us to work on individual stories as a team if there were stories that we were having trouble on.
- We also decided to make Slack updates mandatory every other day. This helped in making sure that we kept each other updated on the stories that we were working on. Implementing the mandatory slack updates gave us time to update each other on stories that we weren't going to be able to complete in advance, so that another team member could work on the story if they had time.

- During the second half of working on the project, we began to communicate and work together a lot better. Through our communication, we developed better friendships to be able to work more efficiently as a team, be able to solve problems quicker, and communicate with each other with a more straight-forward attitude that identified the issues clearly and quickly without anyone taking that attitude personally. I think that was a huge obstacle that we overcame that allowed us to be successful and efficient as a team.

- Troubles that you ended up solving or finding.

- We had a limit on our API keys so we had to cycle through different keys so we could keep testing/developing.
- When we first implemented the car search dropdowns we made an API call for ‘model’ every time the user changed the ‘make’ and ‘year’ fields. This slowed down the interface drastically. We fixed the lag by placing all the makes, models, and years in a text file. Placing the make, model, and year in a text file eliminated the lag and used less API calls.
- The issue of lack of and/or bad communication in the beginning that caused our progress to slow down due to some occurrences of uncomfortability with bringing up issues that were present. After we overcame this obstacle and we all came to be comfortable with each other, our progress and efficiency increased because issues were brought up immediately, and no one had a problem doing so because of the relationships that we came to build.
- The research that came with the new library we were using. New functions and elements to build the window that we had not used before and setting them up so that we displayed what we wanted when it came to color patterns, sizing, padding, etc.

(8) Things To Be Improved (Also Retrospective)

- Parts of the software that you would improve

- Use different programming language because Java doesn’t allow for a sleek and beautiful user interface unlike JavaScript or Python

- Design a better user interface; one that doesn’t lag when moving between windows and doing the API calls. Also one that looks a little more modern, smooth, and attractive that will attract users.
- Web app instead of desktop app. A web app would've looked much better and performance would’ve been faster.
- Improve personality matching so that the cars returned match the personality more often than not.
- Car calculations. There weren’t any set formulas to calculate what we needed, so we had to use parts we found in our research and include some parts of our own to make it as accurate as possible. Even then, the calculations are only about 80%-90% accurate (they can be more accurate, but these percentages are on the low end of the accuracy and were given to be on the safe side.)

- Parts of your teamwork/process that you would improve on in the future

- Have more communications between the team, everyone should get on slack to report their status updates almost everyday so the whole team knows each of its members progress.
- We should have started meeting on thursdays so we could work on the project much earlier. Meeting face to face is a lot better than texting over slack or even a google hangout.
- Make sure everyone knows how to use Github and be able to push, pull, commit, and merge changes
- Put more effort into sprint planning and break down individual stories so the team is more likely to finish their stories

(9) Lessons Learned

- Advice you have for future COMP 129 students

- Keep everything organized from the start so that everyone is one the same page as to what is done, what is currently being worked on, and what still needs to be done to complete the project.
- DON'T OVER COMMIT! After you are done with sprint planning, whatever you commit to the first time, take that commitment and remove a couple stories, because you will ALWAYS over-commit no matter what you think. You can always start more stories that you didn't commit to from the product backlog.
- Don't just work on what you're comfortable with. Plan and allow everyone to work on all parts of the project. Even though it might seem like it will be harder and take longer, it will benefit you a lot more because you will learn and gain experience in a variety of things. Not only that, but the team will benefit because everyone will know how to do every part of the project. This way if a team member needs to leave or can't work on the project a week, the team won't be affected because there will be other team members that can take over their part.
- Break down your stories. After you think you're done revising, revise again and break them down again. The smaller the stories, the easier they'll be, the quicker you'll get them done, and the easier you'll be able to understand what it is that you need to do. This will also help you to know exactly when you are complete with a certain feature.
- Communicate! Make sure you keep your team members updated on whatever messaging client you're using, in our case, Slack. Update team members on stories as you're working on them and when you finish them. Also, let your team members know before the end of the sprint, if you're going to be able to finish the stories assigned to you. This way another team member has the opportunity to work on the stories and can possibly complete them in time for the sprint.