COLORADO SCHOOL OF MINES
EARTH • ENERGY • ENVIRONMENT

# CSCI 370 Final Report

**S A G E**

**S**ydney John **A**nthony Anosov **G**riffin Rutherford **E**than Stacy


CSCI 370 Summer 2022

Prof. Donna Bodeau

Table 1: Revision history

| Revision | Date | Comments |
|---|---|---|
| Rev – 1 | May 26, 2022 | Transferring content from separate documents onto this template. Adding in missing sections from requirements and design. |
| Rev – 2 | Jun 10, 2022 | Transferring content from additional documents turned in throughout the last few weeks for field session. Filling in future work, lessons learned, and project completion status. |
| Rev – 3 | Jun 12, 2022 | Updating previous section to accommodate changes in the project's design |
| Rev – 4 | Jun 13, 2022 | Final sections filled in and more key terms added |
| Rev – 5 | Jun 14, 2022 | Report review suggestions added |

# Table of Contents

# I. Introduction

Uber began as a ridesharing company based in San Francisco, connecting drivers to passengers to help people get around town. Uber is about enabling movement to people's everyday lives. Since then, it has expanded to food delivery with Uber Eats and Postmates, package delivery, and now freighting. Uber Freight is a subsidiary of Uber which helps connect freight drivers to shipping companies. Uber Freight aims to empower and modernize the freighting industry by offering drivers and shippers better transparency, technology, and opportunities to get on the road to deliver goods.

Our objective is to design and implement a proof-of-concept of a self-documenting automation engine which, given an arbitrary set of processes and inputs for them, can execute and document the inputs and outputs of the set of processes. Traditionally, bugs in an engine are exceptionally hard to pinpoint because the operations performed by the engine are not self-documenting. This problem forces software engineers to look through much more code than is necessary to find and fix a bug. A self-documenting engine provides end users such as business owners and inventory managers with more transparency into the automation processes driving their business. Documenting processes in a human-readable way makes it easy for those without a computer science background to take matters into their own hands and investigate the sources causing undesirable outputs. Once the sources are identified, they can be quickly relayed to the engineers.

# II. Functional Requirements

The engine must be able to accommodate a variety of different contexts and serve as a conduit for function calls. This engine is not limited to just Uber. Other companies and even users at home could make use of it to document their method calls. For example, such an engine could invoke and document processes such as scheduling appointments for a freighting company, solving complex mathematical equations, or determining whether a patient's blood tests show unhealthy levels of cholesterol. To accomplish this, the engine must be built with a high enough level of abstraction to not make any direct references to the codebase it is working in tandem with. In practice, this means the engine must not make instances of classes from the program it is working with.

Class method signatures can vary greatly (static vs non-static methods, method overriding, variable parameters, different return types, etc.). The engine must still be able to accommodate for such differences without hard coding in advance. The engine must also handle cases where the output of a process must be used as the input to another. Many larger programs have processes that communicate with one another and are seldom one-dimensional. Not accounting for this would make the engine viable only for a niche type of program.

While our engine does not need to be optimized for performance, it cannot compromise the functionality or overall performance of the software that it is being used on. Due to the limited time constraint and the limited knowledge of the team, adding in parallel processing capabilities is also not necessary.

The engine must compile the documentation of each process listed in an input payload, which can be easily exported and presented in a human-readable form. It is important that those without a software engineering background can investigate problems revealed in the documentation on their own, without having to entirely

rely on a team of software engineers to find problems. This is to also make the lives of those engineers easier since they would not have to sift through the entirety of their code to find one problem.

## III. Non-Functional Requirements

The engine must be packaged in such a way that users can easily add any of their programs into the library's class path. This can be done by refactoring the engine into a separate module and putting external programs in their own module as well. The build files can be updated to import the engine into the external program and vice versa, where it can then utilize its functionality.

The development of the engine will serve as the pilot project for the proof-of-concept report requested by our client at Uber Freight. The report must be written in an academic style and address whether creating such an engine is feasible within the given time constraint. It is important to note that the engine will not be used by Uber Freight in any way, shape, or form, and the team is not receiving any compensation from Uber. This is strictly a learning opportunity for the team to work on something more akin to a real-world project. The engine must also not be written in the Go programming language in order to reduce the chances that an employee at Uber can drop the code into Uber's production code.

## IV. System Architecture

A fully realized engine would belong in between a company's data center and external API calls, as shown in Figure 1. The data center handles normal business logic. API calls come in to request information from the data center, potentially requiring a calculation to be done. The engine can act as the mediator between both the data center and the API call; it executes that which is specified in the call and interacts with the data center to complete the task. Our engine documents the inputs, name of the process, and outputs of any decision-making process along the way. The engine then compiles all the documentation and displays it to the end-user.

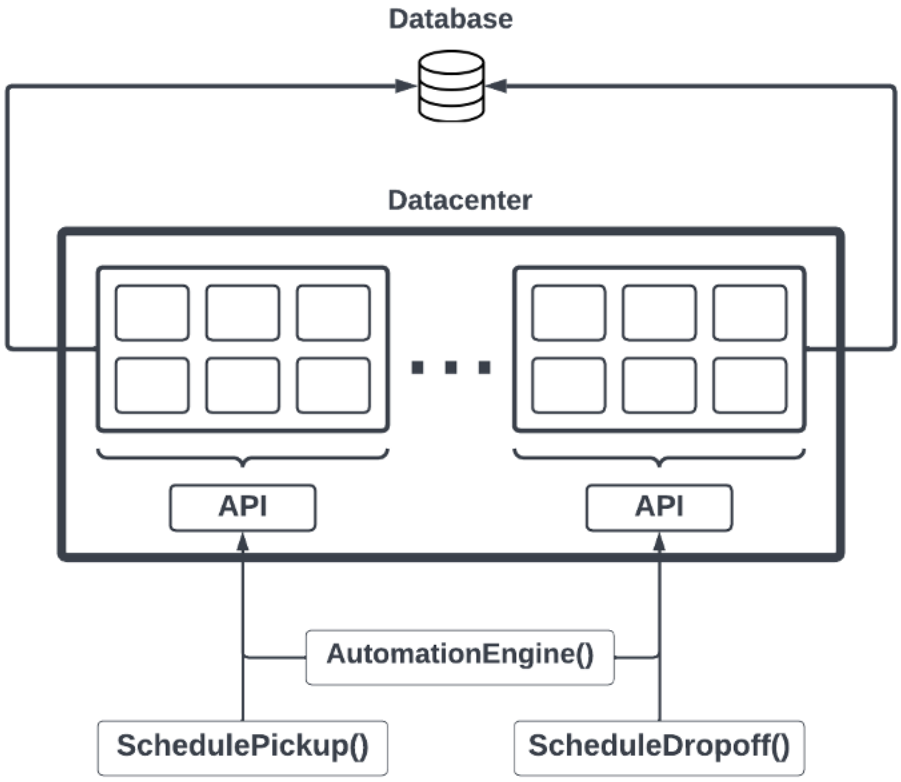


*Figure 1: Example engine implementation in a data center environment*

We are using JSON payloads to feed target parameters and inputs to the engine and output documentation to the user. For this, we designed a JSON parser built from the Jackson library (a JSON processing Java library). Jackson will help extract all the necessary information from the JSON payload that the engine needs in order to invoke the proper methods, as well as supply them with the correct inputs. The payload will be parsed and sent to a map which contains the necessary methods needing documentation. The engine populates the methods with their parameters and executes them. While executing, the engine is documenting the processes being executed and their respective inputs and outputs. Finally, the engine will send the documentation to a JSON compiler, which will compile the documentation into a payload containing human-readable JSON. Figure 2 is an example of the type of abstraction the engine is trying to achieve when documenting a method. If the input to a factorial method is 5, the user should not worry about how it's achieving an answer of 120 (in other words, showing 5 * 4 * 3 * 2 * 1 is not necessary).

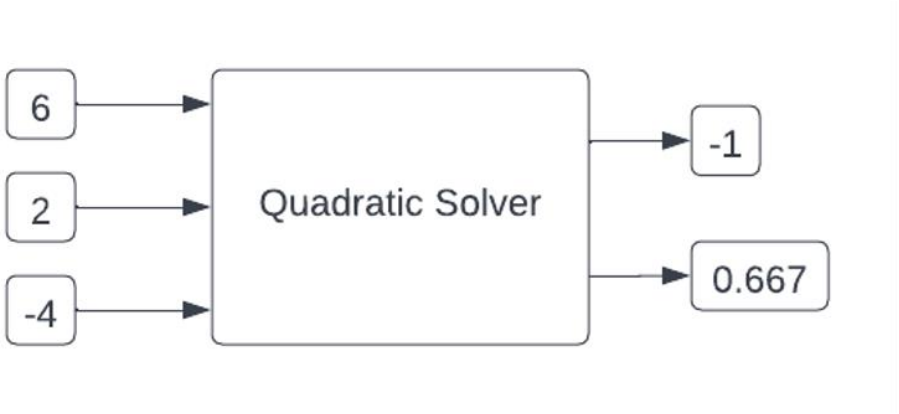


*Figure 2: Concrete example visuals of a process' attributes needing documentation*

Our Engine is centered around a class conveniently named "Engine". The "Engine" class is a driver class that compiles all the necessary components of the program and runs them. It begins by creating an instance of a JSON parser, which parses an input JSON file and stores the information in an immutable record called "TaskDef". The information from the record populates a class called "RunContext", which controls the flow of the program and serves as the brains of the program. It passes on the necessary information to a class called "Runner", which processes methods and their parameters. After the runner has compiled both the parameters and the methods in a way that can be invoked abstractly, the runner then makes a new instance of a "Documentor", which performs the method invocations and documents the output. That documented method gets sent back to the "RunContext" class, where it gets added to the master documentation. Finally, when there are no more methods to be executed, the Engine grabs the documentation from "Documentor" and outputs it to the command line, as well as compiles it to an output JSON file. Figure 3 is a flow diagram showing the high-level overview of the program and the components relative to each other.

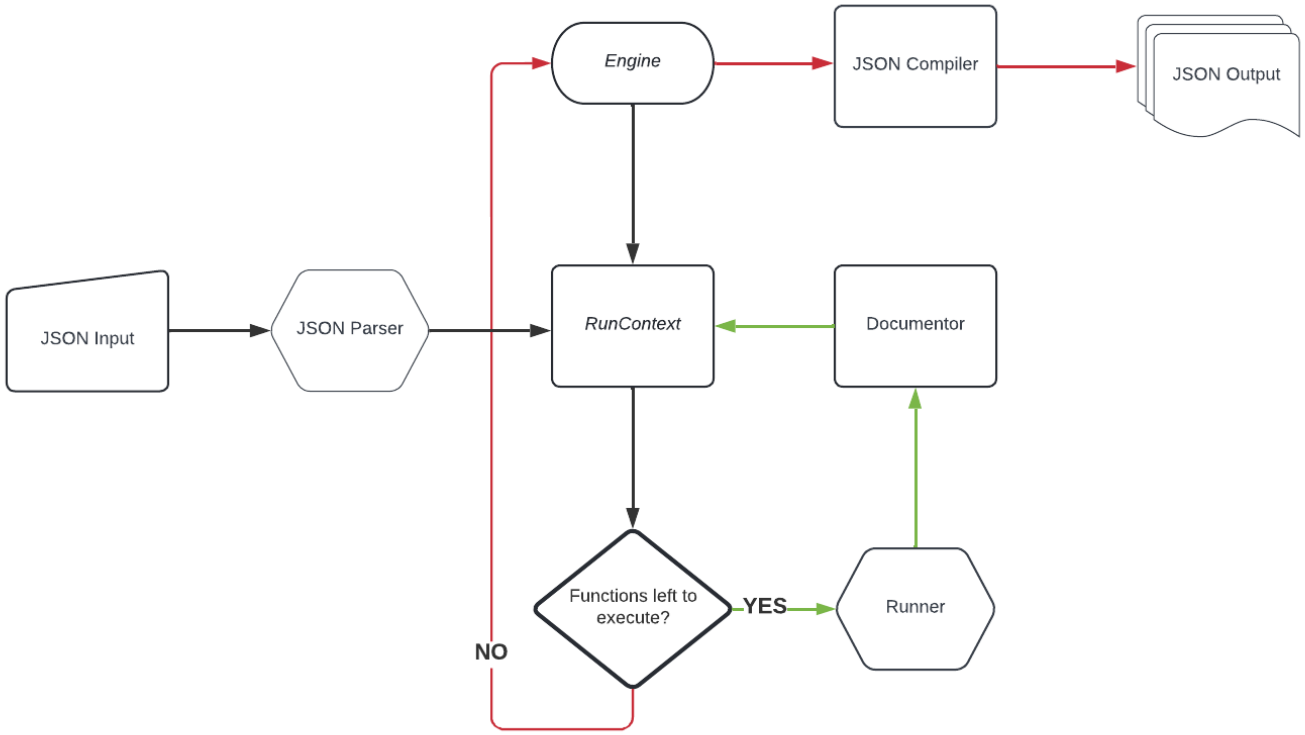


*Figure 3: Concrete example visuals of a process' attributes needing documentation*

The team used Java Reflections API for some of the core functionality in the "Runner". Reflections is a way to examine and modify entities during runtime through the Java Virtual Machine (JVM). Using it, we can examine classes and their fields, parameters, methods, and constructors at runtime, without directly making a reference to any of them in the engine. Methods can even be invoked without directly calling them. However, there are limitations to it. Certain things like package, class, and method names are needed to be defined in the input JSON file for Reflections to cleanly invoke methods. Attempting to do so without those inputs can add a lot of unnecessary complexity to the program.

Java Stream API was also used in tandem with Java Reflections. Streams are able to perform processing on collections, arrays, or I/O channels and provide results from pipelining methods. Streams do not change a collection's structure, rather offer a more programmatic approach to pipelining results through intermediate operations. Figure 4 shows how the "Runner" class is using reflections to gather the methods from a class using "funcClass.getMethods()", using streams to filter them by the name specified in the input JSON file, filtering them again by parameter lengths, and mapping to transform it into a new instance of "Documentor", where the method will be invoked and recorded. Both "m.getName()" and "m.getParameters()" are also examples of reflections being used in figure 4.

```
private List<Documentor> getMethods(RunContext context, Class<?> funcClass) {
    return Arrays.stream(funcClass.getMethods()) Stream<Method>
            .filter(m -> functionDef.functionName.equals(m.getName()))
            .filter(m -> m.getParameters().length == functionDef.params.size())
            .map(m -> new Documentor(context, m)) Stream<Documentor>
            .toList();
}
```

*Figure 4: Use of streams and reflections in the "Runner" class to get methods needing invocation*

## V. Software Test and Quality

To ensure the consistent quality of the engine, a wide variety of unit testing and refactoring were used. It was necessary to prioritize the engine's core functionality itself versus trying to make its output aesthetically pleasing. UI testing was omitted from the development process as a result. While the team did not utilize TDD (test-driven development), unit tests were written to ensure that the core functionality of the engine remains functional as features were continually being added. Additionally, other components of the program (including the JSON parser and compiler) have tests which check for successful parsing of a variety of different inputs. Figure 5 shows an example of a small unit test for the JSON parser, checking to see that the "factorial" field in the JSON file was parsed correctly and contained the correct value.

```
@Test
public void parseJsonInt() throws JsonProcessingException {
    JsonParser parser = new JsonParser(jsonTest);
    int ans = parser.getKeyValueAsType("factorial");
    assertEquals(5, ans);
}
```

*Figure 5: Example of a unit test testing for correct parsing of the factorial field from the JSON input file*

Functional testing was crucial for checking whether the engine would work with other unrelated programs. Several arbitrary classes were developed that contain a variety of processes to execute (math functions, appointment schedulers, etc.). These classes simulate separate programs which import the engine and use it to invoke and document their methods. The intent is to make the engine into a library that users can import and use, regardless of what their programs are about. Thus, this portion of the testing process was especially prioritized.

The GitHub repository that used for collaboration and version control was also configured to automatically build the code and run the tests for every commit, showing an error and sending an email message if it fails. This

consistent and automatic checking incentivizes only working code to be pushed, preventing extra debugging work.

Heavy refactoring has been done throughout the program to make the code as concise and efficient as possible. Local variables were extracted from complicated lines, and appropriate naming conventions were enforced. Some scenarios that were tested with JUnit include methods that depended on other methods, exception handling for incorrect formatting for input JSONs, and ensuring that the output JSON is formatted consistently (especially with nested method calls).

Code quality was important for the team. Refactoring was done often in order to compartmentalize and encapsulate chunks of functionality. This helped greatly when adding more functionality to the program and made navigating the engine easier as complexity grew. It is important that the code is well refactored and contains sound coding conventions in case a different team begins further development of this engine in the future. Libraries such as Jackson were used to help keep code more concise. Java Reflections and Stream API helped with avoiding writing too much messy code for the abstract portions of the engine, as well as make complex parts of the engine more readable.

Our client regularly provided feedback on possible coding strategies which gives us additional ideas to work from to create the most robust engine we can. This was especially useful for helping the team get unstuck when the abstract nature of the program became hard to digest. The team also utilized help from mentors and colleagues with more experience to give suggestions on design decision and libraries that could be used in order to create a more sophisticated engine.

## VI. Project Ethical Considerations

The engine itself does not have any specific ethical concerns, but issues could arrive when the engine is working in tandem with another program. The inputs and outputs of a program's invoked methods are logged to a JSON file, which if not encrypted could reveal classified information about a company's business logic without their consent. This issue is particular to applications that deal with confidential information like passwords, medical data, or social security information. Additionally, a malicious actor could use the engine as malware and infiltrate a company's data center, where incoming and outgoing API calls can be logged, as well as processes occurring inside of the data center.

The engine currently does not have any inherent encryption techniques being used on the documentation. This is because different products being developed using this library could demand a large variety different encryption standards. It is up to the client to implement their own security features to ensure that information does not get leaked. The team highly encourages users to take the initiative to protect their documentation.

## VII. Results

While we are able to fully implement a prototype engine in Java and Python, this did not include a graphical user interface. Doing so was a stretch goal, and the JSON output files could be visualized quickly using the many JSON visualizers available online. Additionally, the Java version of the engine is not able to handle incorrect

JSON formatting or other types of edge cases. In other words, the engine assumes an intelligent user who will configure the input file correctly. An example of an edge case is a user not inputting the right number of parameters for a method in the JSON file. Users can also choose to input non-matching data types, like inputting an integer as a string. Finally, many programs use unconventional data types. JSON does not have a "Date" data type. Users would mostly likely have to input it as a string, which the program would have to convert. In a real-world situation the input JSON would likely be created by another computer system, ensuring correct formatting. Both the Java version of the engine and the Python version have been fully implemented as libraries that can be run with other programs.

In terms of testing results, the final version of the engine is able to pass all unit tests relating to parsing, compiling, and processing correctly formatted JSON data. Each of the smaller components and their unit tests remain functioning. Most importantly, functional testing went successfully. We made two separate projects that were in an external module. These programs were added to the class path of our library so that a jar file can be executed with the input JSON file passed in as an argument. The engine successfully handles two programs containing completely different problems without having to change code inside of the engine. Besides the lack of custom exception handling and dealing with the many corner cases, the core functionality is present.

## VIII. Future Work

A lot of the base functionality of the engine has been completed, but there is still a lot of room for future improvement. As mentioned earlier, the engine assumes that the user configuring the input JSON file is intelligent and does not make any mistakes. If, for example, the user specifies a certain function with two parameters to be executed but only lists one parameter in the parameters field, there are currently no custom exceptions that handle the problem. There are a large variety of ways a user can incorrectly configure the file, and the team did not have enough time to create custom exceptions for them. Creating custom exception handling in the program would be a good first place to start if this engine gets picked up by another team.

The engine currently has a lot of flexibility regarding static vs non-static methods, varying number of parameters, and method return types. It is a little trickier to deal with the varying parameter-types of methods being invoked (like the "Date" example from the previous section). Many development teams implement a custom converter class or library to address this issue. This can be a very tedious process and it would have taken away a lot of time from developing the rest of the engine. Since the team had gotten behind at some points throughout the project, the time constraint applied a lot of pressure and certain features could not be implemented.

The engine uses a JSON input; this was very often reiterated throughout this report. In contrast, many of the most popular programs out there can accommodate a large variety of files. Perhaps it would be worth investing the time to implement a CSV file parser, for example. Another idea is to allow Groovy DSL files to configure the engine in Java. Because Groovy is a language developed on JDK (Java Development Kit) and runs on JVM, it is very compatible with Java code and can configure the engine using code rather than JSON. This is very appealing, because it would make the engine more powerful by providing it dynamic inputs which assist the engine in executing tasks. It would also alleviate some of the parsing work that the engine must do. Unfortunately, this approach is not very user friendly and would only serve software engineers.

Finally, the documentation is presented to the user in a JSON format. While JSON is human-readable, it would be better if graphical interfaces were developed later to make the documentation even more presentable. This is especially appealing to those who need their processes documented so that they can show them to important company stakeholders. There are several third-party options such as PlantUML, which offer several diagraming tools including JSON visualizers. The documentation could also be presented in other file formats besides JSON, such as a simple text file.

## IX. Lessons Learned

In an open-ended project like this, it is necessary to maintain consistent communication with the client. On a minimum of a weekly basis, the client should be informed of the status of the project, and the team should be proactive to ensure their progress fits the needs of the client, which could change depending on the progress made by the team, as well as the time constraints throughout the field session.

Additionally, it is important to keep in mind the limitations of different languages when trying to implement a system like this. Our prototype engine was created in both Python and Java which exposed the limitations of an older, statically typed language. Compared to Python, Java requires vastly more work and creative design in order to achieve feature parody. The biggest limitation in Java is that functions/methods cannot be passed as arguments, restricting our ability to use straightforward solutions such as decorators and requiring the client to put in more work to define what our engine should do.

Throughout the field session, the team learned several techniques to achieve the abstraction needed to implement such an engine. Firstly, the team explored Java Reflections API, which is a feature in Java which allows a program to inspect and obtain internal properties of itself during runtime. In order words, information such as class names, fields, methods, parameters, and more could be obtained at runtime without making direct references to them. This is especially helpful for compiling the necessary parameters to a method and invoking said method without hard coding it in the program.

The engine often works with collections, whether it's a collection of parameters or a collection of methods. Advanced processing of these collections was done using the Java Stream API, which is a feature which helps process elements programmatically. Stream API behaves in a more pipelined manner compared to collection processing through a data structure living in memory. While we did not delve too deeply into learning Stream API, it certainly appears in our code and plays a crucial role in dealing with invoking methods of different parameter types and number of parameters.

To properly document invoked methods in earlier versions of the engine, the team opted to create a wrapper class which wrap around method invocations and record the inputs, the method name, and the output. To accomplish this, the team learned about several common OOP design patterns and chose the one which suits best. While a decorator design pattern was ultimately chosen, the proxy and chain-of-responsibility design patterns were also researched as potential alternatives in case the decorator did not function the way it was intended to.

# X. Team Profile

**Anthony Anosov**
Senior
Computer Science
Hometown: Chicago, IL
Experience: SAP CX and CRM Solutions (SAP Hybris)
Interests: Astronomy, Space Exploration, Lifting, Hiking, MMA, Sleeping

**Ethan Stacy**
Junior
Computer Science
Hometown: Denver, CO
Interests: Robotics, Snowboarding, 3D Printing, Graphics Programming, Astronomy, Sleeping

**Sydney John**
Senior
Computer Science: General Track
Hometown: Zillah, WA
Work Experience: Medical Health Assistant, PHGN200 TA
Interests: Lifting, Hiking, Programming Languages, Reading

**Griffin Rutherford**
Junior
Computer Science: Data Science Track
Places Lived: Plano, TX; San Salvador, El Salvador; Santa Fe, NM; Golden, CO
Work Experience: CSCI101 TA, Calculus Tutoring, NM State Park Trail Guide and Office Clerk
Interests: Exploring mountains, Singing, Snowboarding, Psychology, Lifting, Old Nintendo Games

# Appendix A – Key Terms

| Term | Definition |
|---|---|
| *Engine* | *A system that provides the core functionality of a program (in this case, our program)* |
| *API* | *Stands for Application Programming Interface, which provides a service between two or more pieces of software* |
| *JSON* | *JavaScript Object Notation, a standard file format for representing human-readable data* |
| *Business Logic* | *Foundational code that does "what the business is about". Features that do not directly interact with the customer are not part of business logic.* |
| *Java Reflections API* | *A feature in the Java programming language which allows a program to introspect in examine properties about itself at runtime* |
| *Java Stream API* | *A sequence of objects with a variety of supported methods which are pipelined in order to produce a desired output* |
| *JVM* | *Stands for Java Virtual Machine, which processes and executes Java bytecode in a portable way* |
| *JDK* | *Stands for Java Development Kit, and is one of the core packages used in Java programming, and contains useful tools for developers to use when writing Java code* |
| *DSL Files* | *DSL stands for domain-specific language. DSL files are used to configure Gradle, our project's build tool* |