Salesforce Team 1:
Test Dependency Analysis

Team: Adrian Estrada, Bao Nguyen, Joss Chapman, Torin Johnson, Willie Ruemmele

Client: Salesforce, Matt Buland

June 11th, 2019

Who is Salesforce:

Salesforce is the world's largest CRM (Customer Relationship Management) company, with 35,000 employees across 52 locations worldwide. They were the originators of the Software as a Service (SaaS) business model, and one of the first adopters of 'The Cloud'. They were founded in 1999, and have managed to transform their business into a multibillion dollar company, with 2018 having a total revenue of 13.3 billion dollars.

What is the Vision:

Phase 1: In this portion of the project, our client wanted us to write a program which creates a spreadsheet to show function and class dependencies. To do this, we took in Aura files (Salesforce specific code file) and JavaScript files. We parsed these files for functions, and algorithmically found the dependencies. Finally, these dependencies have been sorted and exported as a .csv file. We call this phase the Code-to-Code Phase due to its nature of finding dependencies between pieces of code.

Phase 2 (Stretch): Using the created dependency file, the next step is to create an application that will help the developer identify which tests to run based on the changes to the code base that they make. We call this phase the Test-to-Code Phase due to its nature of connecting tests to instances of code. It should be noted that this phase is partially a stretch goal, with the requirement that we begin working on it without the expectation of finishing.

Phase 3 (Stretch): Using the test-function/class associations, we will reverse the process, making it so any test case can determine what functions are associated with it, therefore streamlining the manual testing process by showing a developer how tests and functions/classes are associated. We call this phase the Code-to-Test Phase due to its nature of reversing the Phase 2 process. It should be noted that this entire phase is a stretch goal, with the client having no expectation of us making any progress on it.

Functional Requirements:

The functional requirements for a project are the specific inputs and outputs that are to be expected from our program. Below is a brief overview of our listed functional requirements for each phase of our program. We have formatted this in a list, as that was how the requirements were given to us.

Phase 1:

- Output a .csv spreadsheet which shows code dependencies
- Output linked files
- The program must take in Aura and JavaScript files

- The program must be easily usable by the command line

Phase 2:

- The program determines tests based on code dependencies based on the functions in the code

Phase 3:

- The program determines tests based on code dependencies based on the tests chosen

Non-Functional Requirements:

The non-functional requirements for a project include the design components, as well as any aspect that isn't a direct input output requirement designated by a client. In other words, the components of our project which we get to control and change dependent upon what we are most comfortable with doing (the how of the project).

Phase 1:

- Parses JavaScript and Aura files
- Is able to classify functions in JavaScript and Aura (lexing)
- Is able to classify function calls in JavaScript and Aura
- Is able to classify class/library dependencies
- Is able to determine code dependencies
- Is able to format a .csv file using dependencies

Phase 2 (Stretch):

- Is able to classify which tests are to be run based on the Aura and JavaScript code

Phase 3 (Stretch):

- Is able to classify which Aura and Javascript files go with each test, based on the test itself

System Architecture:

Diagrams:

All Phases Flowchart:

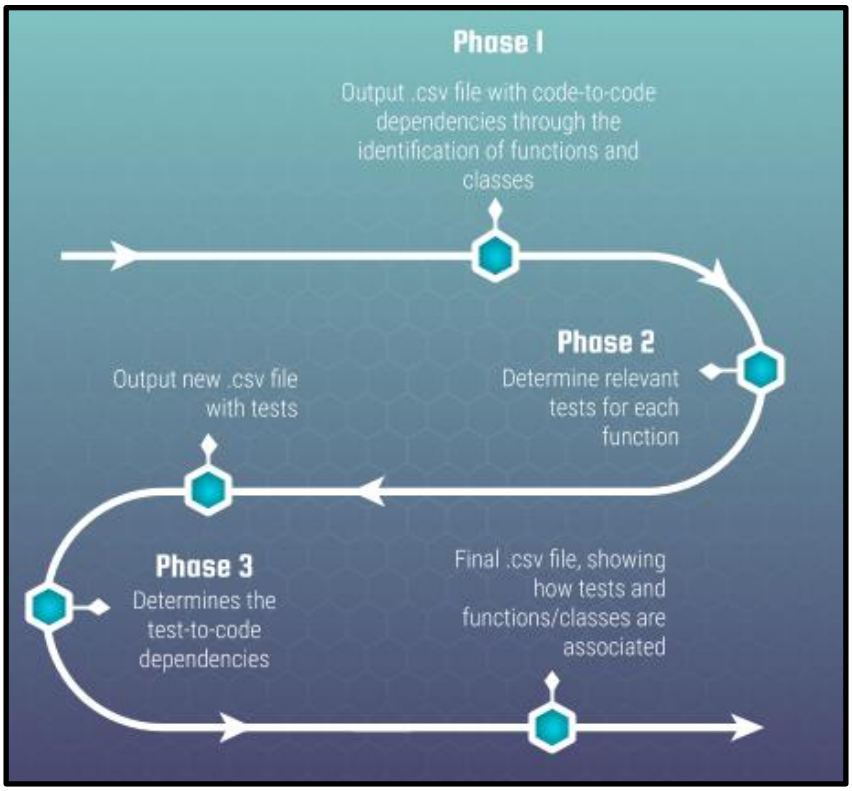

Figure 1: The above flowchart runs through our programs full structure with the included stretch goals (phase 2 and 3). It is a simplistic model to help with a basic understanding of how our program works (once fully completed) on a surface level, as well as the direction we move during the creation of it.

Program Input/ Output Diagram

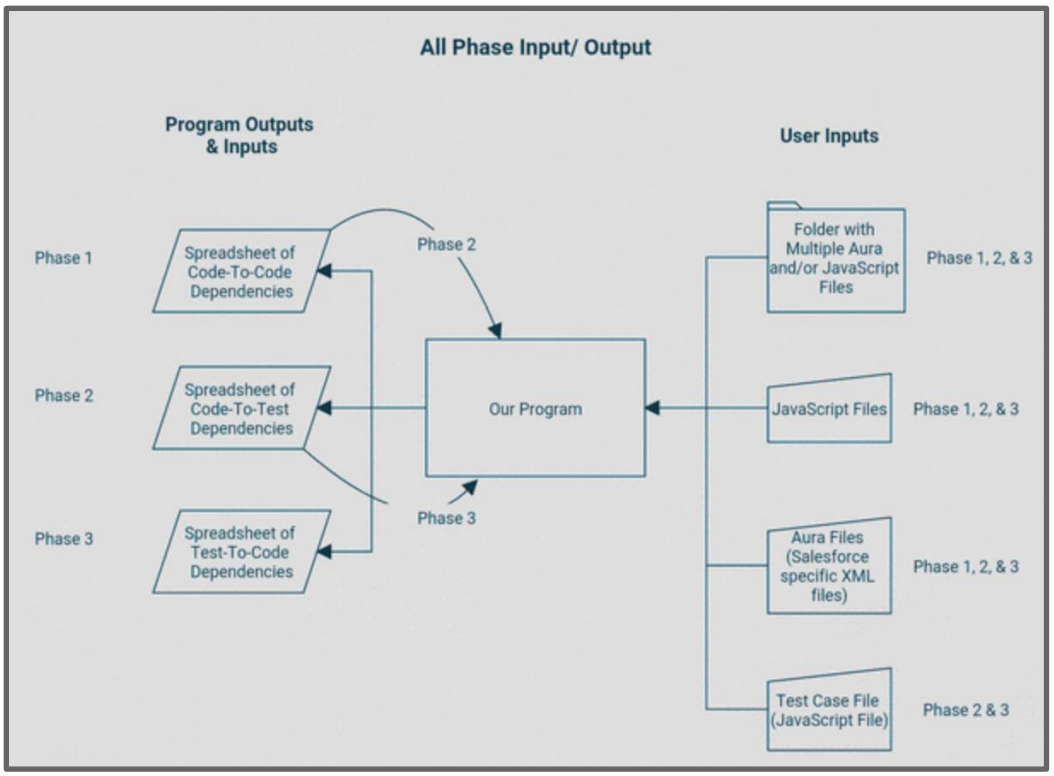

Figure 2: The above diagram shows the inputs and outputs of our program. Next to each input or output is the phase in which it occurs. In addition, some outputs are passed back as inputs in other phases (denoted by a curved arrow). The other flowchart and UML contain more information on the inputs and outputs.

Technical Design:

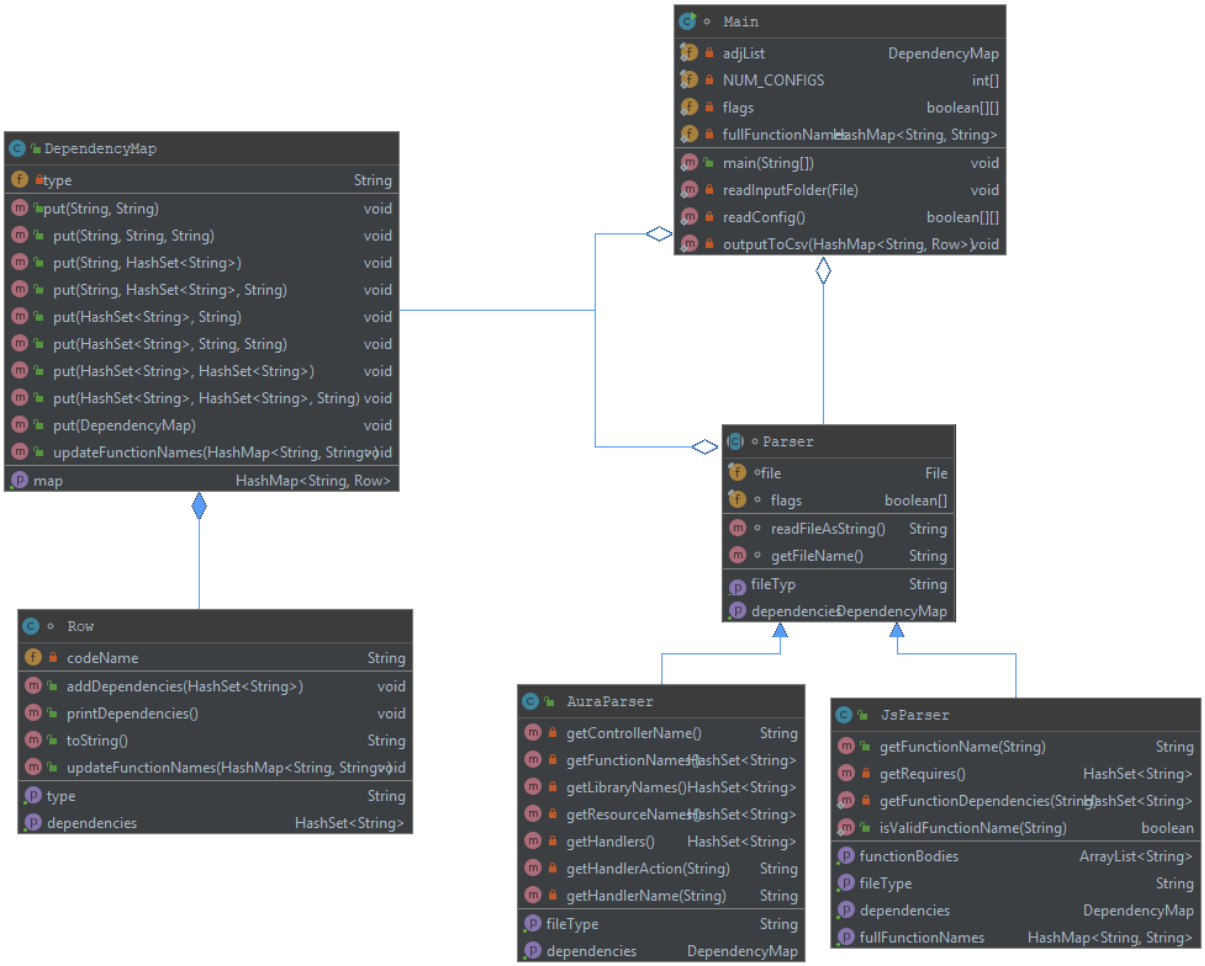

Figure 3: The above UML is associated with Phase 1 (Code-To-Code) of our project. The Main class uses Parser objects. Files are read into the Main class, which are passed into the different Parser subclasses (depending on the file extension type). The Parser classes then parse the code, and search for specific dependencies (function, controller, class, etc.). A DepedencyMap (which is made of Row objects) is then returned to the analyzer class to be used in the creation of a spreadsheet file for the user.

DependencyMap

The purpose of the program is to find all of the dependencies between different types of files. In order to keep track of these dependencies, we created a DependencyMap class. The DependencyMap works similarly to a Map<String,Set<String>> (an adjacency list), but with some additional functionality.

The most important functional difference is the very overloaded put() method. This method is how you add new dependencies to the map. However, there are several

different possible ways that you may want to add dependencies, depending on what type of file you're parsing, and which direction you consider the dependency to go. The most straight forward is when one item depends on a set of items (e.g. a JavaScript function depends on all of the functions that it calls). Also pretty straightforward is when one item depends on another item. You can also use put() for when a set of items all depend on a single item (e.g. all of the resources listed in navEventManager depend on it). There's even a version for if all of the items in one set depend on all of the items in another set (e.g. All of the functions in a JavaScript file may depend on all of that file's imports). Most of the versions of put() end up calling some other version of put(). When adding dependencies, the DepenencyMap identifies if they are for items that it already has dependencies for, and correctly combines the keys and values.

Each DependencyMap contains a single HashMap. While using a HashMap from Strings to Sets of Strings may be easiest, this would only work if the only data needed for the output were the names of the items (the String) and a list of items they depended on (the Set of Strings). Because the output requires additional data for each item, a slightly more complex data structure was required. We created a Row class. Each Row object contains all of the data needed for a single Row of the output, including the name of the item, its type, and, of course, a set of its dependencies. Each DependencyMap contains a HashMap from Strings to Rows. (This does result in a bit of redundancy, because the key in the map is the name of the item, and the value, a Row object, also contains the name of the item, but it was necessary to have that data in both places so we could access the correct Row).

When a new DependencyMap is created, you can give it a default type. This means that whenever it creates a new Row, it will give it that type. This is very convenient because most items identified by a Parser will have the same type (e.g. most of the dependencies identified by an AuraParser will be for Aura Components). However, there are certain instances, mostly due to bidirectional dependencies, where a Parser will identify a dependency for an item with a different type. In this case, a third argument can be used in the put() method, specifying the type. For this reason, for each of the four possible pairs of inputs (String-String, String-Set, Set-String, and Set-Set), there are two versions of put(), one that takes a type String and creates rows with that, and one that just uses the default type for that DependencyMap.

The most important DependencyMap used by the program is the one in the main class. This one mostly uses a version of put() that simply takes another DependencyMap as an argument, and then adds or combines any dependencies from the new map. Unlike the DependencyMaps created in the Parsers, it does not have a default type, so all of its Rows just have a type from the ones made in the Parser objects.

Finally, both the DependencyMap and Row classes contain an updateFunctionNames class. When a parser identifies a call to a JavaScript function, it has no way to determine the class or namespace that contains that function. However, as each JS file is parsed, the program learns which functions it contains. By the end, it builds up a large Map identifying which files contain which functions. This basically gives it a dictionary, mapping the short name of the function (used in the function call) to the full name of the function including the namespace and class names (expected in the output spreadsheet). After running all of the Parsers and agglomerating the full DependencyMap, it calls the updateFunctionNames() method, passing it this dictionary. The DepenencyMap, as well as all of the Rows that it consists of, then use this dictionary to update the keys and values of their own maps with the full function names. Since the full function names are all consistent, dependencies from multiple Parsers which actually refer to the same items are combined, which results in an extensive, but non-redundant description of the dependencies in the code base.

dependencies.config

One of the most frequent problems that we ran into while developing the program was that “dependency” is subjective, and the details of it can be very subtle. For example, does an Aura Component depend on the JavaScript Functions that it calls, or do the Functions depend on the Components to call them correctly? Or both? To deal with this question, we made the program configurable. A dependencies.config file allows the user to decide the details of what dependencies should be included in the output. This meant that our client didn’t have to conclusively decide some of the finer details of dependency while we were developing. Just by changing an ‘N’ to a ‘Y’ they can change their mind at any time to decide what works best for them.

```
Aura Component depends on JS Function that it calls: Y
JS Function depends on Aura Component that calls it: N
Aura Component depends on Aura Component that it lists as resource: N
Aura Component depends on Aura Component that lists it as resource: Y
Aura Component depends on Library: Y
Library depends on Aura Component: N
Aura Component depends on Handler Action: Y
Handler Actions depends on Aura Component: N
-------------------------------
JS Function depends on JS Function that it calls: Y
JS Function depends on JS Function that calls it: N
JS Function depends on Required File: N
Required File depends on JS Function: N
```

Figure 4: The dependencies.config file

The program reads the data (the Y’s and N’s) from the dependencies.config file into a two-dimensional, non-rectangular, boolean array. The size and shape of the array are based on the number of types of Parsers and the number of configurable settings

for each type of Parser. (When a new Parser is initialized, the appropriate array of settings is passed to the constructor.) By setting it up like this, the user can add more types of Parsers later on, or add more configurable settings to an existing type of Parser, and just have to update one number. They don't have to worry about it throwing off other settings. This is an example of our program adhering to the open-close principle.

The text before the colon on each line is ignored by the program. We put in text that we believe clarifies what each setting controls, but because the program doesn't care about it the client can write whatever they think is the best explanation, and optimize the file for them and their coworkers.

While we tried to make the reading of the config file as robust as possible (empty lines, lines starting with a dash, anything before a colon, and extra lines at the bottom are all ignored, and the Y's and N's are case-insensitive), there were a few things that could not be ignored. This includes a missing config file, a config file with not enough lines, or lines that don't include a setting after the colon. While it would be easy to just set any unknown values to a default, this felt like it could lead to problems. There would be no way for the user to know that dependencies they want included are being left out because the program isn't telling them that their config file is wrong. To avoid this problem, if any of those errors are detected the program outputs a clear error message and terminates.

Parser

The heart of our program just comes to our parser class, and the children it has spawned, which hold the actual functionality. The path we decided that would be best for finding dependencies in the code was through different regular expressions. We started off by figuring how a function is defined in both Aura and JavaScript, and then moving on to finding out how these functions are, and can be implemented. Finding all of these forms and implementing them into the form of a regular expression was the key to being able to actually find the function names when traversing the files. After these functions that belonged to other classes were found within files, we were then able to put the function names into a map, where the file they were found within would be the key. We found this would be the best way to keep track of all dependencies within a file, as the key could be a file, and the values would be the functions found within the file that were not defined in that file. This was our way of implementing the parser's output to ensure this would return a spreadsheet like our client had requested. This made everything easier in the long run, as the map key and values could each be put in respective columns, making the spreadsheet readable, and not having too many rows for just a single file.

Software Quality Plan

As the quote by Alan Lakein goes, “Failing to plan is planning to fail”. There is no exception within the creation of great software, and as such our team has a software quality assurance plan. This covers not only how we wrote the code on a technical level, but how our team functioned to assure we were running efficiently and effectively. Through the below sections, our client can be certain that the code they received will be of the highest quality we can provide.

Team Ideation

Before every meeting, we had the entire group do a scrum meeting to discuss what we’ve done, what we plan to do, and what issues we have with getting there. After this, we usually sat in a semi circle around a whiteboard, and went more in depth about how we plan to accomplish our plans for the day with each other. These plans included coding, writing team documents and other documentation, and structuring and practicing presentations. We laid out what we would do, and how we should go about it to everyone, so we were each given room to speak our minds and then receive feedback on these plans. We were also clear on who would be working on what initially, and mention how we would switch up roles throughout the day to ensure everyone will have a hand in everything. This makes sure as much ideas were being spread as possible, which allows for better solutions to be formulated and applied.

Client Check-ins

While our team initially was unable to make contact with our client due to him having a surgery, we were able to keep him up to date with our moves constantly in the second half of field session. This allowed him to give us feedback on how we were doing, and updated requirements he may have for us. This has led to our team having confidence in ensuring that we have met the client’s needs. He also had access to our git repository, so he always had access to where we were currently at, allowing him to demo our program with whatever tests he had whenever he pleased. Whenever we had questions, we made a list and emailed them to him, and he was usually very quick on responding, so it allowed us to not remain stuck for long, and we were able to move forward with finishing our product.

Pair Programming + Code Reviews

Essentially all programming done for this project was completed in pairs. This was to ensure ideas are able to be thrown around to ensure a more well rounded deliverable, and someone will always be able to catch the other’s mistakes. We also had constant code reviews, as two people were normally working on the code at one time, so we often had others look at the code, and make suggestions. We often would swap members out for coding during this time to ensure that everyone had a hand in creating everything.

Programming Structure

To ensure our code is up to the quality standards required, we made sure to use SOLID principles within our code. Specifically, the open closed principle and single responsibility principle were often in discussion when we were moving on with our code. We have an abstract class for our parser in the code, as we were creating parsers for different languages. This ensures that the client will be able to add or remove parsers, depending on their needs, and each parser is used for their individual language.

Due to our client's surgery, we ensured that extensive documentation went into our code, so we would be able to explain our thought processes and decisions that went into the project. We also make sure to create our comments before our code, so we keep on track with what we're trying to make, and expand on the comments only if we find we were missing something essential.

Programming Testing

Since our program works on the backend of Salesforce, JUnit tests were able to be written for most components of our code. We have used JUnit tests to ensure our code is always functional and ensuring that we do not make modifications to our code, as it would violate our previously created tests. A few specific JUnit tests we've implemented include:

- testControllerName
    - This test ensures that we are getting each controller name correctly. This is done by using a couple known controller names, and making sure they are included in the map that is created when parsing for controller names.
- testFile
    - Ensures there is a file to be parsed.
- testFunctionBodies
    - Makes sure the file has the correct number of function definitions.
- testValidFunctionName
    - Makes sure the function name is a string, and doesn't contain any key words.

Our client has provided us with spreadsheets and multiple examples of Salesforce code, allowing us to have real examples of how our output should look, along with code that would produce these outputs. He also had access to our code himself, so he was also testing our code, and if he ever found anything major, he mentioned he would notify us. This did not happen, as he did not find any major red flags in our program.

Final Results:

The goal of this product was to create a program that finds the code-to-code dependencies between files and folders used by Salesforce. In this, we have succeeded. Right now, we have a working output that shows all code-to-code dependencies in a .csv file, so it can easily be looked at in a spreadsheet. This includes defining what type each dependency is. Initially, our client provided us with a hand-made spreadsheet that showed us what our code should output, and we have managed to automate the process and recreate exactly what was in the example. This is excluding the last two columns of the given spreadsheet, but this is due to the client specifying these contained data that would only be able to added manually. We did this for all code written in JavaScript and Aura, as were the base requirements.

The components we were unable to implement were all stretch goals, which are finding the dependencies from their code to their tests, finding dependencies from tests to their code, and creating a parser for Java. These would be the obvious next steps for continuing the project, with the Java parser most likely being the easiest thing to add. Our client has shown us his work on creating the test to code dependencies, and based on what we have been working with currently, we find this to be the logical next step. Once this step is completed, the output can easily be reversed and used to find the code to test dependencies, essentially finishing two steps at once. The tests we have performed were to ensure that the baseline functions are working, but determining if our final output is correct requires us to look at our output, and compare it to what our client has provided us. With this, we have determined that our code has found all dependencies for the code he has given us. The spreadsheet we were given includes dependencies for code we do not have, which was obviously not found in our spreadsheet, so looking through the file names we were given was required. Other than the stretch goals and requirements we were given, there is not much room for extensions, as the code we are creating is very specific to a certain need by Salesforce, and the specific programming languages they use. The only extensions that would be useful would be to create parsers for other languages that end up being picked up by the company.

From this project, a lot of lessons were learned by everyone when it comes to actual software development. The biggest discovery is how vital communication can be to the development of a product in the way the client desires. While having initial requirements is a good baseline for building the framework of a project, being able to communicate with your client about the specifics allows for development to progress smoothly and without confusion. Another big lesson that was learned is how important using SOLID principles are. The OCP and SUP made it incredibly easy for us to implement parsers for both of the languages we were required, along with making implementation of another parser incredibly easy. We initially had all parsers in one class, but moving on to use an abstract parser class and create individual classes for different functionality we were trying to implement made everything easier.

For this type of project, one that will not be fully finished in the amount of time we are given, we found it was smart to create as much documentation as possible, and ensure the language used will be something that is known by the person or team carrying the torch of our program. This will save a lot of headaches for the next team, and ensures they can start their development off running.

In conclusion, the Salesforce File Dependency project was a success on the basis of the requirements stated from the beginning. We found that being unable to communicate with our client in the first couple of weeks slowed down our work, due to our lack of confidence in the project's direction we were moving it into. But, after being able to make contact again, we made suitable progress, and were able to finish up what we were supposed to. If given a bit more time, we may have had time to implement all the stretch goals, since one stretch phase was so similar to a different one. Overall, this was a project everyone was able to learn from, and we are proud to deliver the code we have come up with.

Appendix:

Using the program:

The following program is run through the command line. The format for running this is java Salesforce1 [folder] [folder] should be a relative or absolute path for the folder containing the files for which you would like to find dependencies. After the program runs, you will be prompted to enter a filename ending in .csv If the file exists it will be overwritten, if it does not it will be created This will contain a table of all dependencies, and should be openable in any spreadsheet reader. Column A will contain the name of a file. Column B will contain the type of file in column A. Column C will contain the file dependencies associated with the file.

#Config File The dependencies.config allows the user to decide the details of what dependencies should be included in the output Empty lines or lines starting with a '-' will be ignored The program will terminate if there are not enough lines, but any extra lines will be ignored The NUM_CONFIGS array in main defines the number of settings in the config file for each type of parser Configurations are determined by the presence of a 'Y' or 'N' after the colon Anything before the colon is ignored by the program

Code Conventions:

We used standard Java coding conventions for our program. This can be assumed to be the case for any class within our code.

Response to Feedback:

We have generally left the introduction unchanged. About half of the feedback we received said our introduction was fine, but one of the reviewers said it needed more

detail, and another said it needed less detail. Since half of the reviewers were okay with it, and the other views were conflicted, we felt it was a safe bet to keep the intro as is.

A lot of the feedback came down to our technical document being a bit confusing, and the tense of some of our paragraphs being different than past tense. We believe we sufficiently resolved these mismatches.

We were also unable to receive permission from a member of the group before it was time to submit, so his views weren't able to be taken under consideration. We do believe we had sufficient feedback from the others though.