MySQL Dependency Extraction

EECS 4314: Advanced Software Engineering
Tabs vs. Spaces
November 20, 2017

Authors

Kevin Arindaeng

Anton Sitkovets

Glib Sitiugin

Nisha Sharma

Varsha Ragavendran

Ayman Abualsunun

Davood Anbarnam

Table of Contents

List of Figures

List of Tables

1.0 Abstract

This report presents an overview of multiple dependency extraction methods, specific to MySQL Server 8.0.2. An introduction and overview of each method is presented. A discussion on how each dependency extraction process works is provided, with use cases to present the interaction between the user and the extraction method. Each dependency extraction method is explained through both a quantitative and qualitative analysis. Lastly, a conclusion and lessons learned is presented based on this comparison process.

2.0 Introduction and Overview

This report presents the findings from performing various dependency extraction techniques on the source code of MySQL 8.0.2. There are various dependency extraction techniques available to reverse engineer large code bases to architectural level. These techniques are mainly used for:

- Analyzing large open source code to optimize software design.
- Easily identifying dependencies and interactions within the code.
- Quickly understanding large or complex code bases, often those with poor documentation.

The report begins with the steps taken to extract dependencies using the three selected techniques:

- Understand
- srcML
- A script (developed by the team)

An example scenario is presented in the form of a use case in this section, illustrating the steps a user would perform in order to extract dependencies of any source code using the script developed by our team. Next, a quantitative analysis of each dependency extraction technique against another is described in detail to get a better understanding on the number of dependencies extracted by the technique. Although all 3 techniques are parsing the same source code, each technique provides varying numbers of dependencies. A qualitative analysis is provided in which a sampling calculator is used to determine the sample size which reflects the target population needed, and to reach to a conclusion regarding these techniques. As well, the noticeable differences between each technique is identified and a rationale is provided. Finally, the potential risks and limitations regarding each dependency extraction technique is examined in detail.

To close out the report, a glossary for various terms and phrases used throughout the document as well as any naming conventions used are presented. The report concludes with the major findings and lessons learned from putting together this document.

3.0 Dependency Extraction

3.1 Extraction using Understand

Understand is a static analysis tool that focuses on source code comprehension, metrics, and standards testing. Its design helps to maintain and understand large amount of legacy code and newly created source code (3rd party and open). It also provides a multi-language, cross-platform and maintenance oriented IDE.[1] Understand helps mainly in dependency extraction. A file can depend on another through import statements, inheritance, implementation, method calls, object initializations and Java annotations. Following are the steps taken to extract the dependencies (refer to Appendix A for pictorial illustration) using Understand:

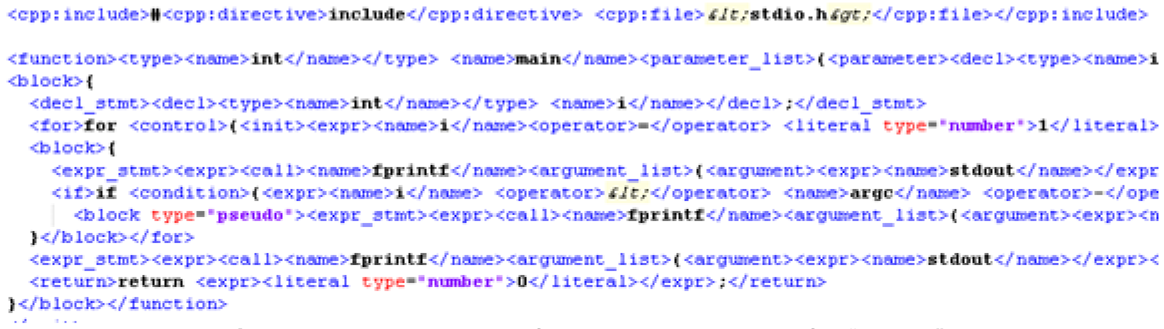
1. Create a ‘New’ ‘Project’ and ‘Name’ it
2. Select the relevant source code ‘Languages’
3. ‘Import’ the source code files (we selected the option to add directories manually)
4. Specify the path of directory with source code
5. Proceed to ‘Analyze’ and ‘Finish’.
6. After analysis is done ‘Report’ is generated for ‘File Dependencies’ and exported to a CSV file.

3.2 Extraction using srcML

srcML is a command line tool that converts source code into a corresponding XML format.[2] To do so, a command that follows this skeleton must be executed:

srcml [directory OR file] –o name.xml

In the case of MySQL, the command *srcml mysql-server-mysql-8.0.2 –o mysql.xml* was executed. The resulting output is best represented by Figure 1, where a snippet of the resulting XML file “echo.xml” is shown after executing the command *srcml echo.c -o echo.xml*:



```
<cpp:include>#<cpp:directive>include</cpp:directive> <cpp:file>&lt;stdio.h&gt;</cpp:file></cpp:include>

<function><type><name>int</name></type> <name>main</name><parameter_list>(<parameter><decl><type><name>i
<block>{
  <decl_stmt><decl><type><name>int</name></type> <name>i</name></decl>;</decl_stmt>
  <for>for <control>(<init><expr><name>i</name><operator>=</operator> <literal type="number">1</literal>
  <block>{
    <expr_stmt><expr><call><name>fprintf</name><argument_list>(<argument><expr><name>stdout</name></expr
    <if>if <condition>(<expr><name>i</name> <operator>&lt;</operator> <name>argc</name> <operator>-</ope
      <block type="pseudo"><expr_stmt><expr><call><name>fprintf</name><argument_list>(<argument><expr><n
  }</block></for>
  <expr_stmt><expr><call><name>fprintf</name><argument_list>(<argument><expr><name>stdout</name></expr><
  <return>return <expr><literal type="number">0</literal></expr>;</return>
}</block></function>
```

Figure 1: Example output from running srcML on file “echo.c”

With the file “mysql.xml” following a similar format to the one shown in Figure 1, the querying language XPath which is built in to srcML was then used to navigate through the XML document to find dependencies. Such dependencies were based on the “include” directive used in C/C++ programming. For example, if fileA was to have the statement “include fileB”, then fileA, referred to as the “From” file, would depend on fileB, referred to as the “To” file. This dependency is thus represented by the following notation: fileA -> fileB. These dependencies were extracted using a three stage process:

<u>Stage 1 – Find all “include” nodes in mysql.xml</u>

This is accomplished using the query: *srcml --xpath "//cpp:include/cpp:file/text()" mysql.xml > mysql_includes.xml*

<u>Stage 2 – For every “include” node, save the “From” file</u>

This is accomplished using the query: *srcml --xpath "string(//src:unit/@filename)" mysql_includes.xml > files_from.txt*

<u>Stage 3 – For every “include” node, save the “To” file</u>

This is accomplished using the query: *srcml --xpath "string(//src:unit/node())" mysql_includes.xml > files_to.txt*

The content of the two files “files_from.txt” and “files_to.txt” were then used to construct a srcML dependency list, where each dependency follows the notation

A -> B. Text processing tools like sed and grep were used to do so, however use of these tools is not an absolute requirement.

The entire extraction process can best be summarized in Figure 2:

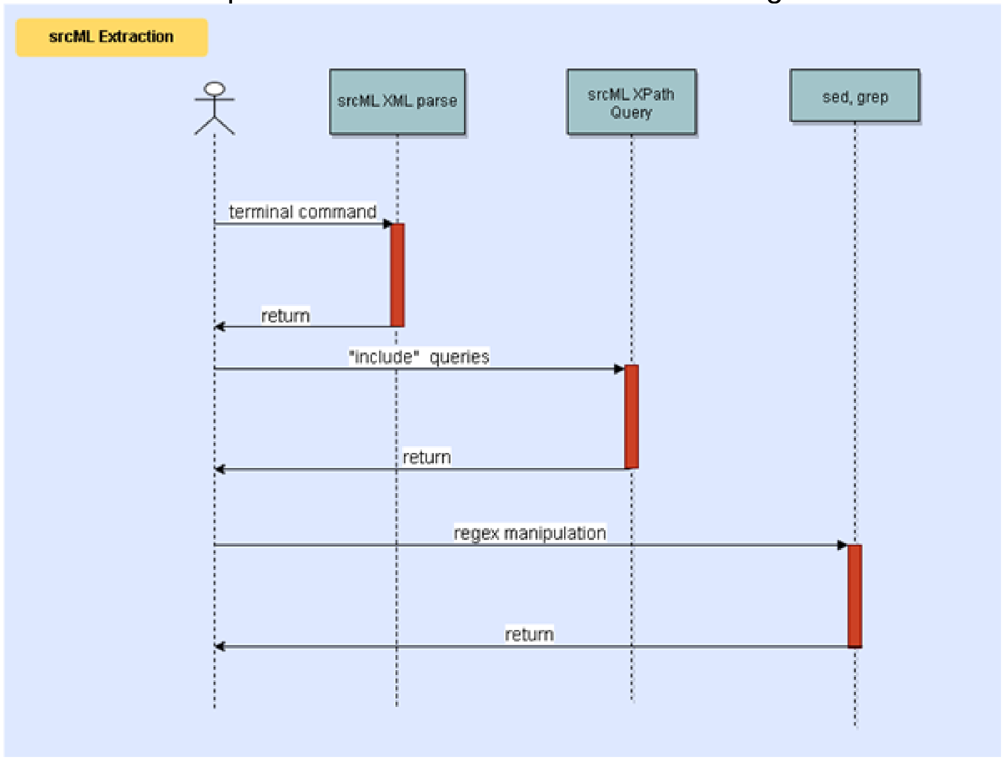


Figure 2: The srcML extraction process

3.3 Extraction using a Personal Script

To examine the difference between Understand and srcML, own script was created that would examine all the code in the MySQL source code and find which files depend on which. This script can be found in Appendix B-1.[3]

This was easily done by walking through the mysql-server-mysql-8.0.2 directory and examining all the .c, .h, .cc, .cpp, .hpp and .java files for their dependencies. It opens the main directory and recursively walks through the directories, while also looking at all the files to check if they are one of the above types. It then extracts the dependencies from this file by looking at it's “#include” directives for C files and “import” statements for Java files.

The next task is to output the result into the ta format, which was done by finding the full path for the file being looked at and then writing the name of the dependency after the ‘include’ or ‘import’. Special care was taken to make sure that when parsing the dependencies. Initially we had trouble with removing unnecessary comments after the include directive and also on how to only extract the name of the Java dependency without all the package information. During our initial creation of this code for the presentation, we accidentally left out ‘.cpp’, ‘.hpp’ and ‘.java’ files, making our comparison results less accurate. Since then we have updated the script to work with these types.

The process for running the script can be described by the following use case diagram in figure 3. First the user must find the directory which includes the script, then from the command line, run: python include.py <path mysql-server-mysql-8.0.2 directory> <output file>.

The one caveat is that the output file must already exist, as the python script will not create the file for you. So an empty file must created initially. The use case diagram describes how the script works by having the user run the script, open and walk through the file directory, while simultaneously looking C, C++ and Java files. Once those files are found, the program will read the files line by line to find any dependencies signified by “#include” or “import”. These dependencies are then written to a user specified output file.

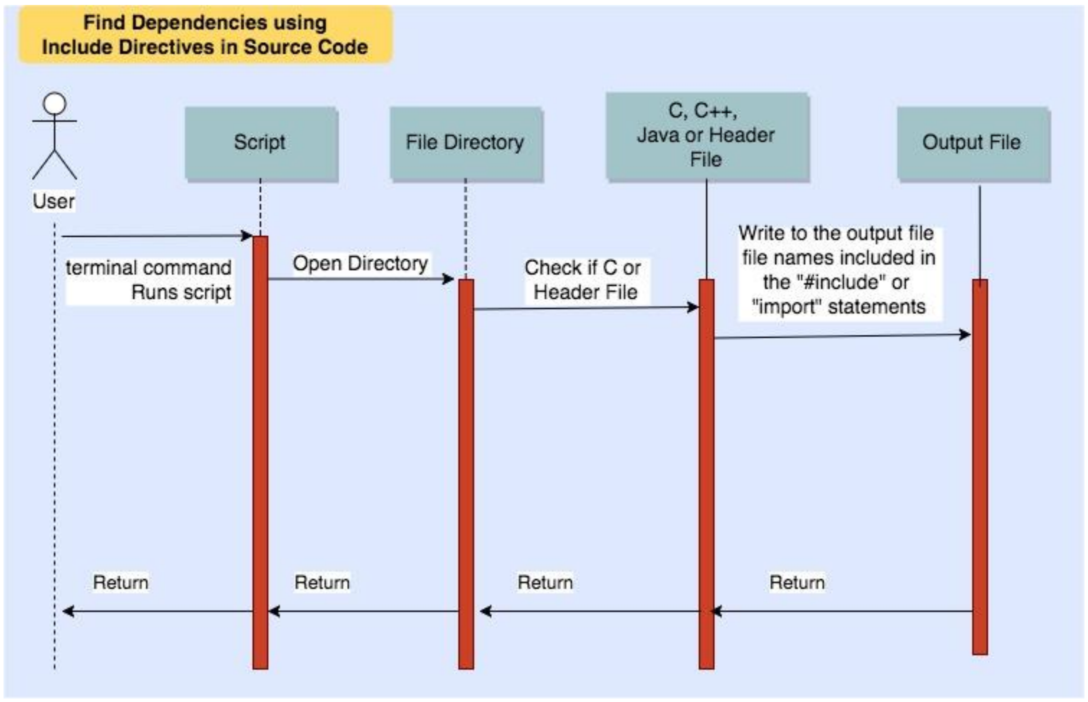

Find Dependencies using Include Directives in Source Code
User
Script
File Directory
C, C++, Java or Header File
Output File
terminal command Runs script
Open Directory
Check if C or Header File
Write to the output file file names included in the "#include" or "import" statements
Return
Return
Return
Return

Figure 3: Use Case of the Personal Script

3.4 Finding Common Dependencies

A simple Java program was initially developed to find common dependencies between the different extraction techniques used, shown in Appendix B-2.[4] The program would take two files as input, and output a user-defined file that listed line-by-line all lines of text found in both input files. While inherently a correct solution, the exhaustive nature of this program results in a quadratic running time, making it a highly undesired and inefficient one that when applied to larger software systems is likely to fail.

Further refinement of the program is shown in Appendix B-3.[5] This eventually led to the use of hashing to reduce the time spent searching for a particular dependency through use of the Java Collection HashSet. This reduces the search for a dependency from linear to constant time, resulting in an improved expected linear runtime for the entire program.

4.0 Quantitative Analysis

4.1 Understand vs srcML

Figure 4 displays a venn diagram that describes the number of dependencies that are common between Understand and srcML.

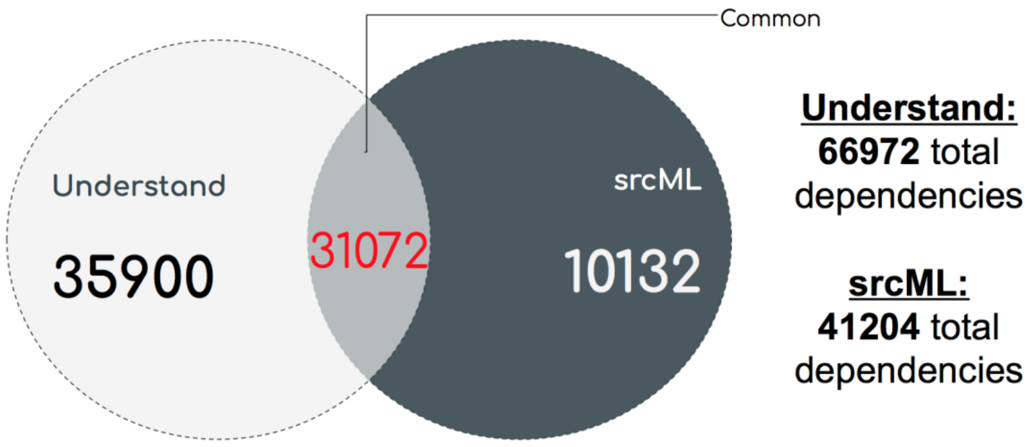


Figure 4: Venn Diagram of Understand vs. srcML

In total, there was 66972 total dependencies extracted using Understand, and 41204 total dependencies extracted using srcML. Understand has approximately 20000 more dependencies than srcML. Using the Java program developed by the team, each dependency extracted using srcML is cross-checked for its existence with all the dependencies extracted using Understand. If the same dependency was extracted by both srcML and Understand, the Java program considers this as a common dependency, and writes this dependency back to a new file containing all the common dependencies.

The number of dependencies in common between Understand and srcML came to 31072 dependencies. Therefore, 35900 extracted dependencies are unique in Understand, and 10132 dependencies are unique in srcML.

4.2 srcML vs Personal Script

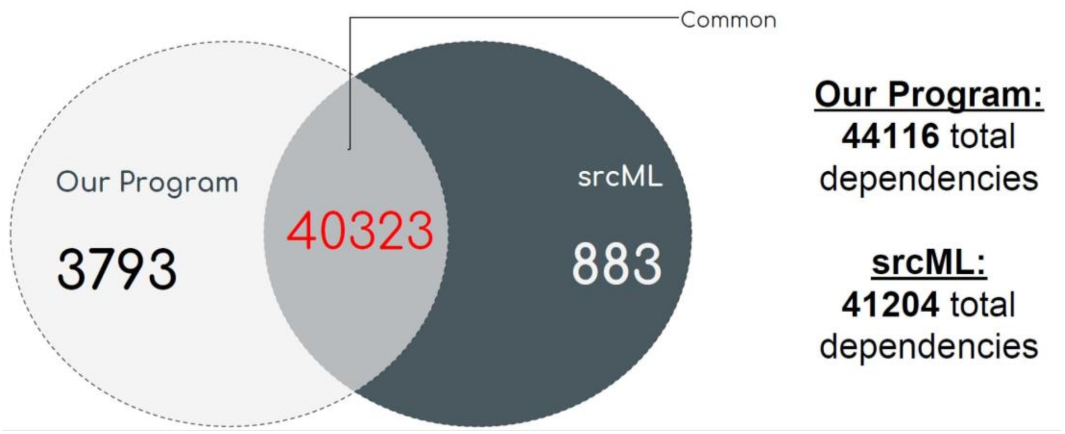


Figure 5: Venn Diagram of our program vs. srcML

In terms of a quantitative analysis of srcML and our program, it was found that our program's data set had 44116 total dependencies, whilst the srcML data set had 41204 total dependencies.

Using a script to detect common dependencies, it was found that our program and srcML had 40323 shared dependencies. 3793 of these dependencies were unique to Our Program, whilst only 883 of these dependencies were unique to srcML. A further discussion on some of the reasons behind this is explained through qualitative analysis.

4.3 Personal Script vs Understand

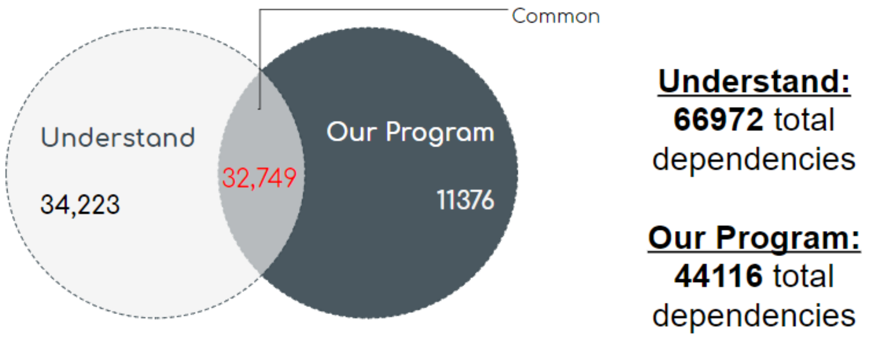


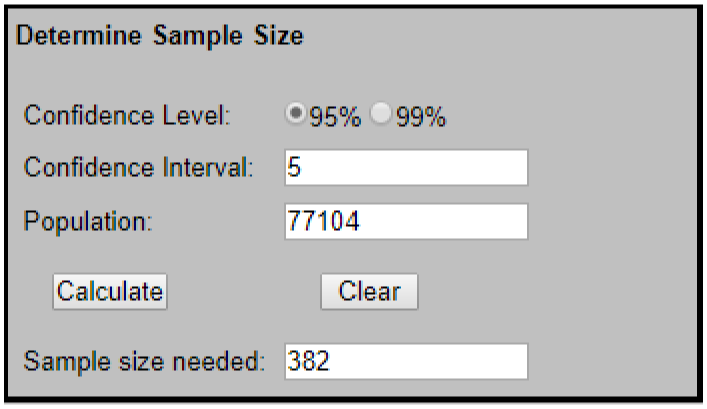
Figure 6: Venn Diagram of our program vs. Understand

Our program detected about 20000 less dependencies than those detected by Understand. The majority of discovered dependencies were shared between Understand and our program (32749). Only 11376 dependencies were unique to our program. The difference between the number of discovered dependencies can be explained by the simplistic approach of our program: the dependencies were only searched in the file headers (include and import statements). As a result, Our program is not capable of discovering dependencies which arise for reasons other than explicit import. Examples of such dependencies would be dependencies which arise due to Inheritance or function calls.

5.0 Qualitative Analysis

5.1 Understand vs srcML

For the qualitative analysis, a stratified sampling method was chosen. Stratification is the process of dividing members of the population into homogeneous subgroups before sampling. Since the given data already divided to 3 subpopulations (common, unique to Understand, unique to srcML), this sampling method was a good fit. A sampling calculator from Creative Research Systems' was used[4], with a confidence level of 95%, and a confidence interval of +/- 5%. with the given data, the sampling size was determined to be 382 as shown in figure 7.


Determine Sample Size

Confidence Level: [✓ 95%] 99%

Confidence Interval: 5

Population: 77104

Sample size needed: 382

Figure 7: Sampling Calculator for Understand vs srcML

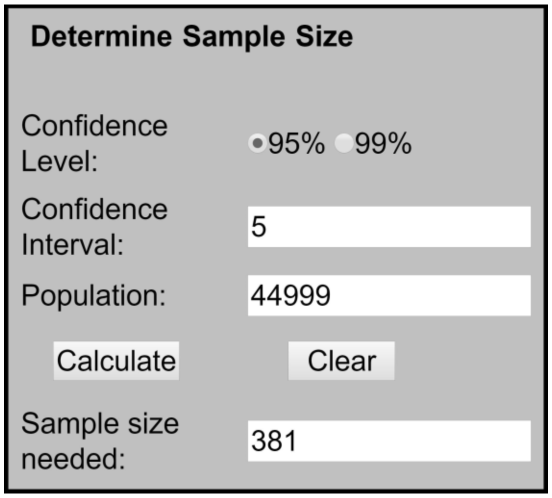
With the sampling size and the 3 subgroups, a stratified sampling table was constructed as shown in table 1.

Subpopulations	**Equivalent Sample Size** (Subpopulation / Total Population)	**Percentage of Total Sample Size**
Common Dependencies	(31,072/77,104) * 382= ~153 cases	(153/382) * 100 = ~40.29%
Understand	(35,900/77,104) * 382= ~177 cases	(32/382) * 100 = ~46.56%
srcML	(10,132/77,104)* 382= ~50 cases	(8/382) * 100 = ~13.14%

Table 1: Subpopulation Sample Sizes for Understand vs. srcML

5.2 srcML vs Personal Script

Instead of looking through the thousands of dependencies between our program and srcML, a stratified sampling method was used to understand the relationship between the common dependencies and unique dependencies. An appropriate sample size for this sampling method was chosen using Creative Research Systems' Sample Size Calculator seen in figure 8.


Determine Sample Size

Confidence Level: [✓ 95%] 99%

Confidence Interval: 5

Population: 44999

Calculate Clear

Sample size needed: 381

Figure 8: Sampling Calculator for srcML vs our program

Here, an arbitrary confidence level and confidence interval (margin of error) was chosen to be 95% and 5 respectively. The population is the total number of dependencies, which includes both unique and common dependencies. The calculator recommends choosing a sample size of 381 to accurately portray the relationship between the shared dependencies of the two methods. The equivalent sample size for each subpopulation is calculated in Table 2.

Subpopulations	**Equivalent Sample Size** (Subpopulation / Total Population)	**Percentage of Total Sample Size**
Common Dependencies	(40,323/44,999) * 380 = ~341 cases	(341/381) * 100 = ~89.50%
Our Program	(3,793/44,999) * 380 = ~32 cases	(32/381) * 100 = ~8.40%
srcML	(883/44,999)* 380 = ~8 cases	(8/381) * 100 = ~2.1%

Table 2: Subpopulation Sample Sizes for our program vs. srcML

Random sampling was used to choose each case, utilizing a random number generator, where the number generated corresponded to the line number of the given data set. To find the unique dependencies for Our Program and srcML, the Unix pattern scanning command, “awk”, was used. A random case from the two minority subpopulations are listed below.

5.2.1 Unique to Our Program

mysql-server-mysql-8.0.2/storage/ndb/clusterj/clusterj-api/src/main/java/com/mysql/clusterj/**Session.java** -> **QueryBuilder.java**

The reason this dependency is here is related to the NDB Cluster independent storage engine and data store. This dependency is part of ClusterJ, a connector that allows users to interface and connect to NDB Cluster using Java. This dependency was unique to our program because it is able find dependencies to both Java and C/C++ files in the source directory of MySQL, whilst srcML was only used to obtain the include directives of C/C++ files.

5.2.2 Unique to srcML

Our program: mysql-server-mysql-8.0.2/rapid/plugin/x/src/**buffering_command_delegate.cc** -> **bind.h**

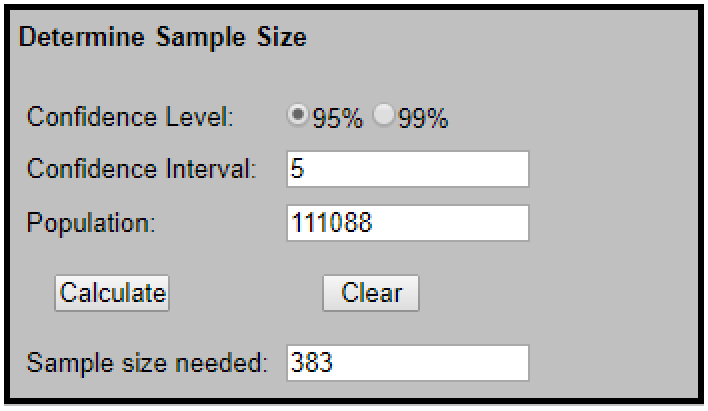
srcML: mysql-server-mysql-8.0.2/rapid/plugin/x/src/**buffering_command_delegate.cc** -> **ngs_bind.h**

Here the dependency found is related to the X Plugin component of MySQL. This is an implementation of a network protocol used to interface with MySQL as a document store. Both of these dependencies were identified by srcML and Our Program. However, due to a minor error in the extraction of srcML, it appended "ngs_" to the prefix of the related dependency. Thus, during comparison of these dependencies, it was flagged as unique for both.

To conclude, the common dependencies between Our Program and srcML share the majority of the total sample size. This is expected as both srcML and Our Program mainly do dependency extraction based on include directives, barring some errors and unique cases.

5.3 Personal Script vs Understand

Stratified sampling method was used to do the qualitative analysis. An appropriate sample size for this sampling method was chosen using Creative Research Systems' Sample Size Calculator seen in Figure 9.


Determine Sample Size

Confidence Level: [● 95%] ○ 99%

Confidence Interval: 5

Population: 111088

Sample size needed: 383

Figure 9: Sampling Calculator for our program vs Understand

To evaluate the qualitative properties of both methods, we used sampling calculator with 95% confidence level and 5% confidence interval. This allowed us to obtain 383 as a recommended sample size.

Then, we used the stratified sampling method to get an estimate of the dependency distribution among Understand and our program. It was estimated that in a sample of 383. About 167 cases would be unique to Understand, while 56 cases would be unique to our program.

Subpopulations	**Equivalent Sample Size** (Subpopulation / Total Population)	**Percentage of Total Sample Size**
Common Dependencies	(32,749/78,348) * 383 = ~160 cases	(160/383) * 100 = ~41.77%
Our Program	(11,376/78,348) * 383 = ~56 cases	(56/383) * 100 = ~14.62%
Understand	(34,223/78,348)* 383 = ~167 cases	(167/383) * 100 = ~43.61%

Table 3: Subpopulation Sample Sizes for our program vs. Understand

6.0 Potential Risks and Limitations

6.1 Understand

Although Understand provides an efficient and effective method of extracting dependencies, it still has some limitations. First of all, Understand is a "black box" software where we can’t get access to the source code, due to which we unable understand the algorithms used in dependency extraction. As a result, we can not reason about the quality of obtained dependencies. Secondly, due to human errors in the process of assigning subsystems to the files an incomplete set of dependencies can lead to incomplete output. Lastly, the method does not let us analyze the whole system, so we need to pick a sample.

6.2 srcML

One of the limitations to srcML is that it appends an extra XML tag before each keyword statement, meaning it requires extra parsing to get the include statements.

6.3 Our Program

The program created mainly looks for import statements and include directives when extracting dependencies. Thus, a limitation would be is that it will miss out on dependencies generated by inheritance, function calls and other deeper dependencies. Another limitation is that the script mainly looks for Java and C/C++ source files. Using this program to analyze other software systems built in different languages will require further changes.

7.0 Data Dictionary

API - A set of subroutine definitions, protocols, and tools for building application software.

CRUD - Create, Read, Update, Delete. Four basic functions of persistent storage.

Dependency - A state in which one object uses a function of another object.

Source Markup Language - A document-oriented XML representation of source code. The purpose of srcML is to provide full access to the source code at the lexical, documentary, structural, and syntactic levels.

SQL - A domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS).

XML - A markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

8.0 Naming Conventions

API:	Application Programming Interface
CRUD:	Create, Read, Update, Delete
GPL:	General Public License
IDE:	Interactive Development Environment
RDBMS:	Relational Database Management System
RDSMS:	Relational Data Stream Management System
srcML:	Source Markup Language
SQL:	The Structured Query Language
XML:	Extensible Markup Language

9.0 Conclusions

This investigation has shown that there is no pre-defined process for extracting dependencies between components in a software system. Each process, however, must define what it means for a particular component to 'depend' on another component, and in that regard similarities can be found between the different approaches used. For instance, the three approaches shown in this document use the "include" directive to define one type of dependency. Such a variety of choice inevitably leads to tradeoffs; one particular approach may be taken due to its simplicity whereas another may be taken due to the in-depth nature of its results, for example. This comparison in particular can be made between the personal script, which relies on "include" and "import" directives, and the Understand tool respectively.

When a choice is made and results are obtained though, the reliability of the data ultimately comes into question. The answer is obtained through comparing results with other dependency extraction techniques, best illustrated in figures 4-6. Inevitably there will be areas common to both techniques, the 'set intersection', and areas unique to both. The former presents no issue as it validates the reliability of the data elements found in that region. The latter, however, signals a need for further investigation, the results of which outline the reasoning for the existence of these regions and ultimately point to differences found in the extraction technique used.

10.0 Lessons Learned

Dependency extraction is inherently an open-ended problem that is best solved using a variety of software tools. In this particular investigation, tools like sed, grep, and in general, shell scripting quickened the process of formatting extracted data to a set standard, while the Java program and the wc command quickened the analysis stage of dependency extraction. As mentioned in section 3.4, the Java program initially used did not present a scalable, efficient solution to the common dependencies problem, signalling a fault in our initial approach to tackling it. To prevent future errors similar to this, a more measured, thoughtful and analytical approach should be taken, where efficiency is strongly taken into account.

11.0 References

[1] Ayaz Isazadeh, Habib Izadkhah, Islam Elgedawy, "Reverse engineering tools," in *Source Code Modularization: Theory and Techniques*, 1st ed.Anonymous Springer International Publishing, 2017, pp. 17.

[2]"srcML", Srcml.org, 2017. [Online]. Available: http://www.srcml.org/. [Accessed: 20- Nov- 2017].

[3] "azkevin/EECS4314", GitHub, 2017. [Online]. Available: https://github.com/azkevin/EECS4314/blob/master/A3/a3data/include.py. [Accessed: 20- Nov- 2017].

[4] "azkevin/EECS4314", GitHub, 2017. [Online]. Available: https://github.com/azkevin/EECS4314/blob/master/A3/a3data/InitialMutualDependnency.java. [Accessed: 20- Nov- 2017].

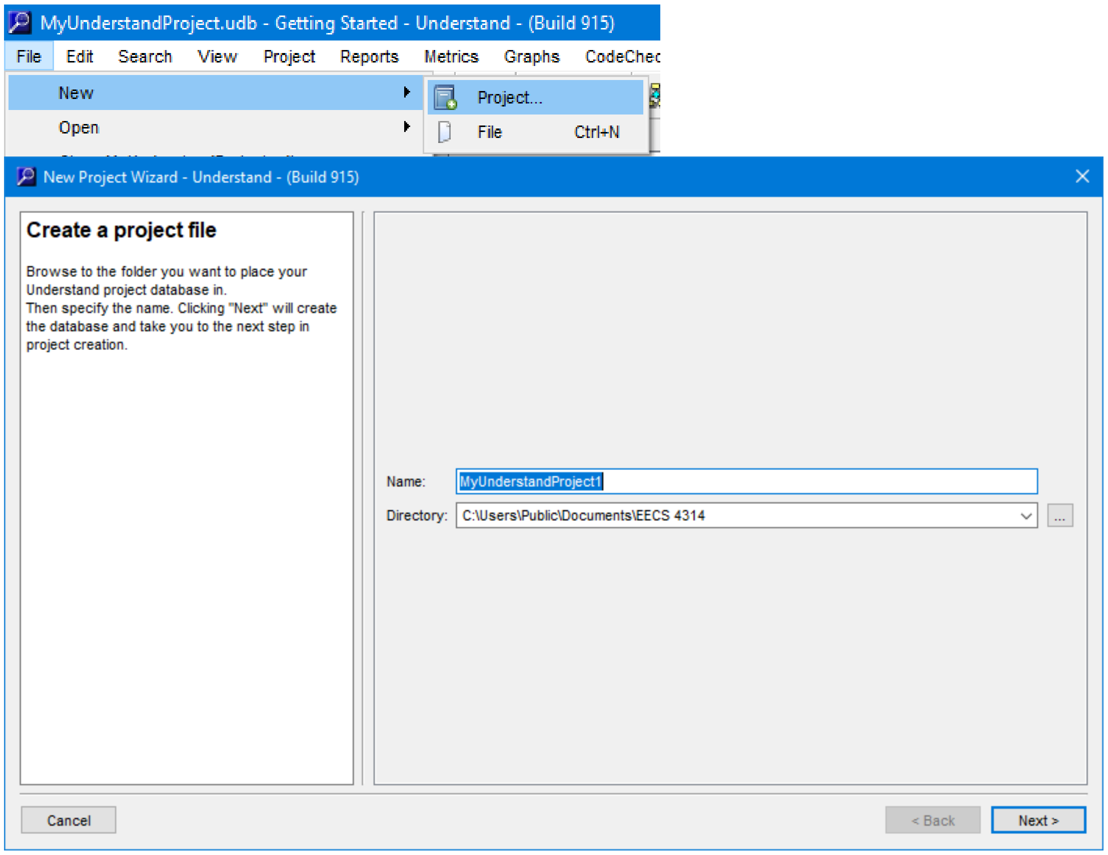
[5] "azkevin/EECS4314", GitHub, 2017. [Online]. Available: https://github.com/azkevin/EECS4314/blob/master/A3/a3data/MutualDependnecy.java. [Accessed: 20- Nov- 2017].

[6] "Sample Size Calculator", Creative Research Systems, 2017. [Online]. Available: https://www.surveysystem.com/sscalc.htm. [Accessed: 20- Nov- 2017].

12.0 Appendix

12.1 Appendix A: Step by Step Extraction using Understand

Step 1: Create a new project and name it


MyUnderstandProject.udb - Getting Started - Understand - (Build 915)
File Edit Search View Project Reports Metrics Graphs CodeChec
New
Open
Project...
File Ctrl+N
New Project Wizard - Understand - (Build 915)
Cancel
< Back
Next >

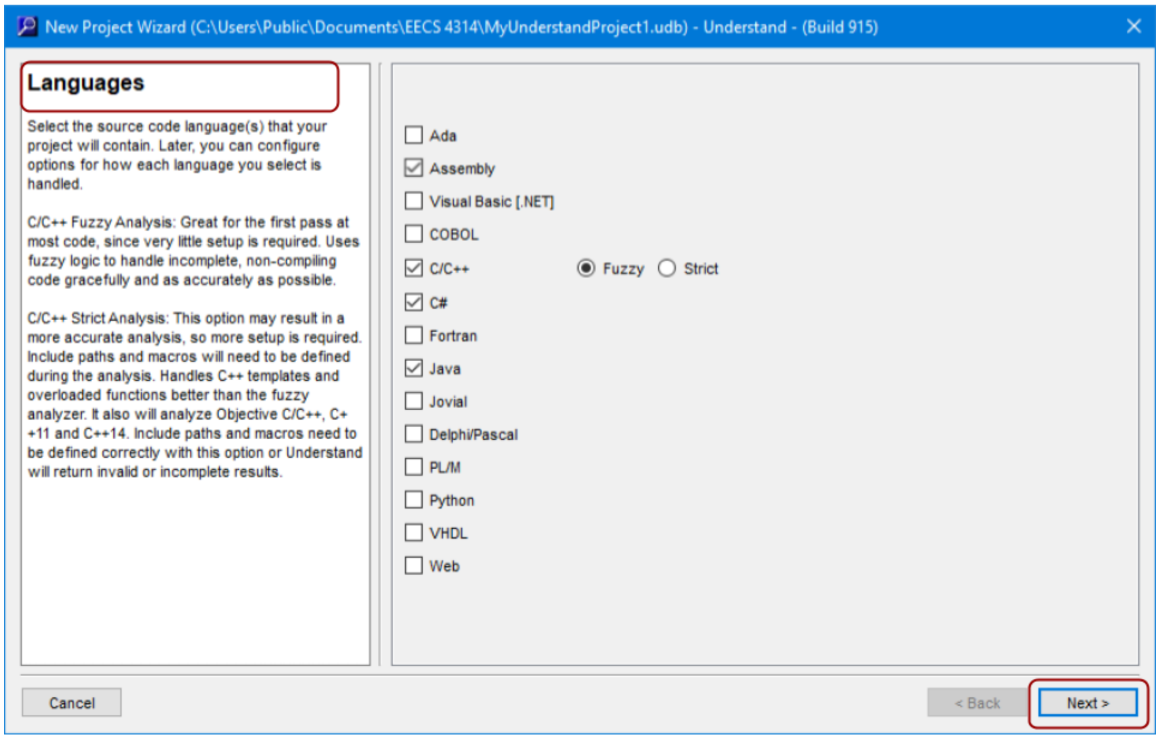
Create a project file

Browse to the folder you want to place your Understand project database in.
Then specify the name. Clicking "Next" will create the database and take you to the next step in project creation.

Name: MyUnderstandProject1

Directory: C:\Users\Public\Documents\EECS 4314

Step 2: Select the Source Code Languages


New Project Wizard (C:\Users\Public\Documents\EECS 4314\MyUnderstandProject1.udb) - Understand - (Build 915)
Cancel
< Back
Next >

Languages

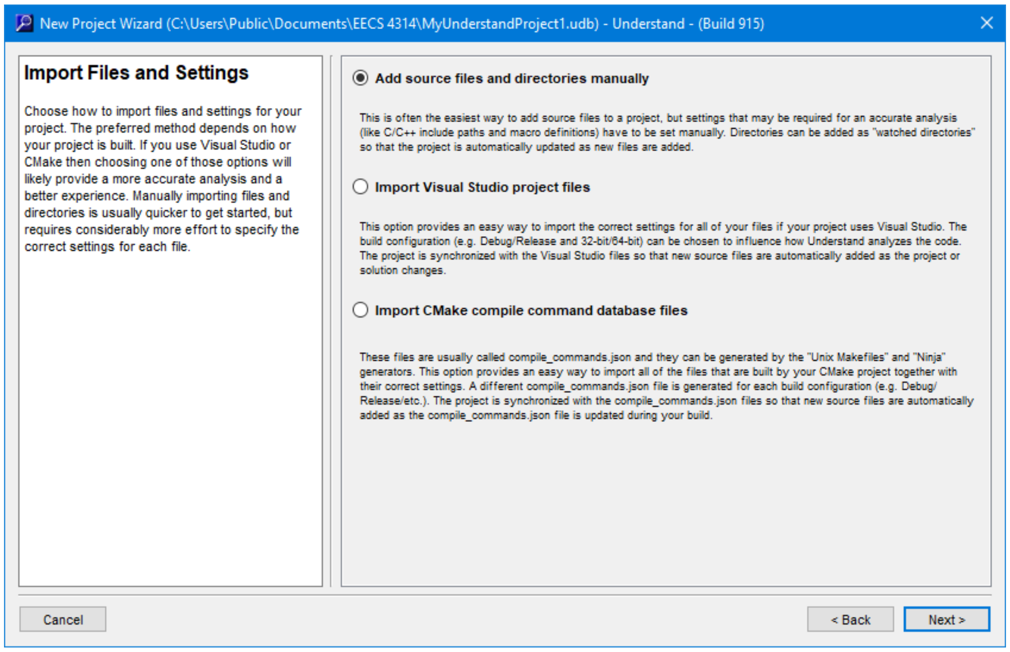
Select the source code language(s) that your project will contain. Later, you can configure options for how each language you select is handled.

C/C++ Fuzzy Analysis: Great for the first pass at most code, since very little setup is required. Uses fuzzy logic to handle incomplete, non-compiling code gracefully and as accurately as possible.

C/C++ Strict Analysis: This option may result in a more accurate analysis, so more setup is required. Include paths and macros will need to be defined during the analysis. Handles C++ templates and overloaded functions better than the fuzzy analyzer. It also will analyze Objective C/C++, C++11 and C++14. Include paths and macros need to be defined correctly with this option or Understand will return invalid or incomplete results.

- ☐ Ada
- ☑ Assembly
- ☐ Visual Basic [.NET]
- ☐ COBOL
- ☑ C/C++ [✓ Fuzzy] Strict
- ☑ C#
- ☐ Fortran
- ☑ Java
- ☐ Jovial
- ☐ Delphi/Pascal
- ☐ PL/M
- ☐ Python
- ☐ VHDL
- ☐ Web

Step 3: Import Files


New Project Wizard (C:\Users\Public\Documents\EECS 4314\MyUnderstandProject1.udb) - Understand - (Build 915)
Cancel
< Back
Next >

Import Files and Settings

Choose how to import files and settings for your project. The preferred method depends on how your project is built. If you use Visual Studio or CMake then choosing one of those options will likely provide a more accurate analysis and a better experience. Manually importing files and directories is usually quicker to get started, but requires considerably more effort to specify the correct settings for each file.

[✓ **Add source files and directories manually**]

This is often the easiest way to add source files to a project, but settings that may be required for an accurate analysis (like C/C++ include paths and macro definitions) have to be set manually. Directories can be added as "watched directories" so that the project is automatically updated as new files are added.

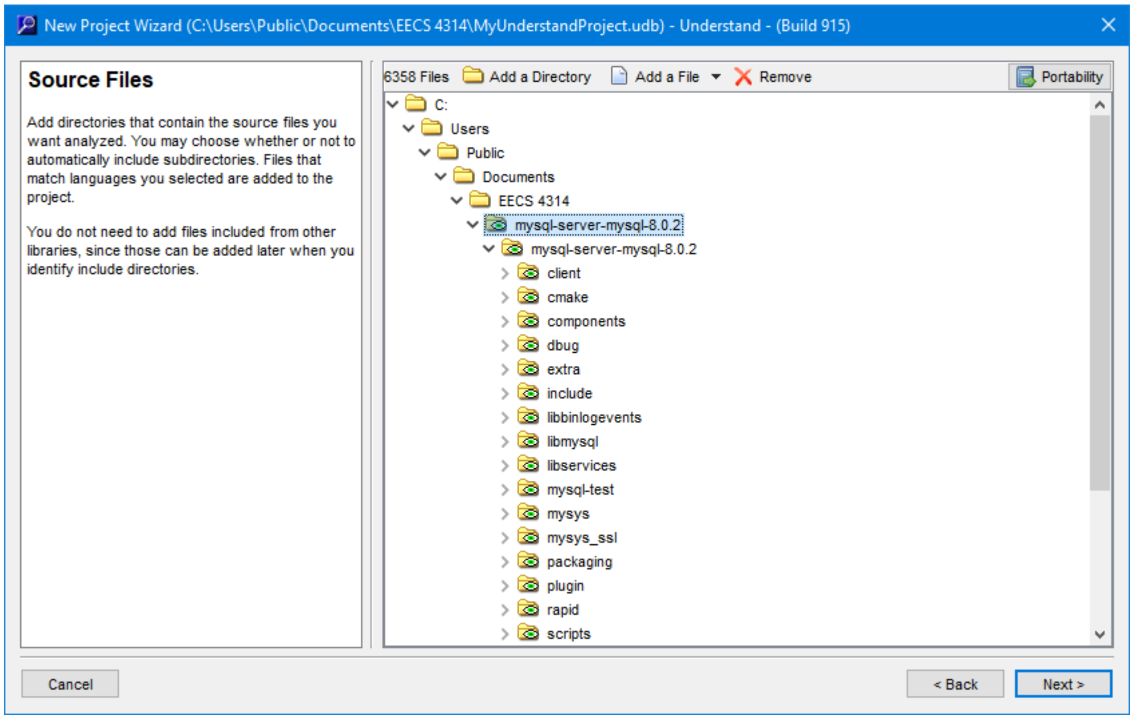
Import Visual Studio project files

This option provides an easy way to import the correct settings for all of your files if your project uses Visual Studio. The build configuration (e.g. Debug/Release and 32-bit/64-bit) can be chosen to influence how Understand analyzes the code. The project is synchronized with the Visual Studio files so that new source files are automatically added as the project or solution changes.

Import CMake compile command database files

These files are usually called compile_commands.json and they can be generated by the "Unix Makefiles" and "Ninja" generators. This option provides an easy way to import all of the files that are built by your CMake project together with their correct settings. A different compile_commands.json file is generated for each build configuration (e.g. Debug/Release/etc.). The project is synchronized with the compile_commands.json files so that new source files are automatically added as the compile_commands.json file is updated during your build.

Step 4: Specify Path of Directory with Source Code



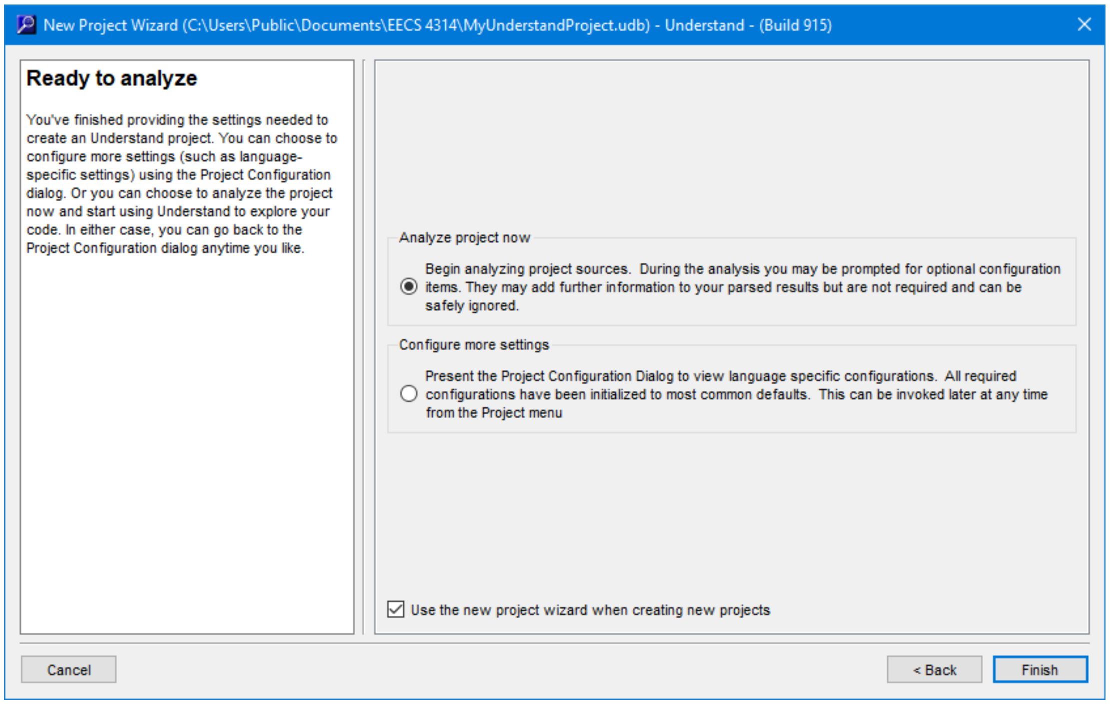
New Project Wizard (C:\Users\Public\Documents\EECS 4314\MyUnderstandProject.udb) - Understand - (Build 915)

Source Files

Add directories that contain the source files you want analyzed. You may choose whether or not to automatically include subdirectories. Files that match languages you selected are added to the project.

You do not need to add files included from other libraries, since those can be added later when you identify include directories.

Step 5: Select Analyze the Project and Finish



New Project Wizard (C:\Users\Public\Documents\EECS 4314\MyUnderstandProject.udb) - Understand - (Build 915)

Ready to analyze

You've finished providing the settings needed to create an Understand project. You can choose to configure more settings (such as language-specific settings) using the Project Configuration dialog. Or you can choose to analyze the project now and start using Understand to explore your code. In either case, you can go back to the Project Configuration dialog anytime you like.

Analyze project now

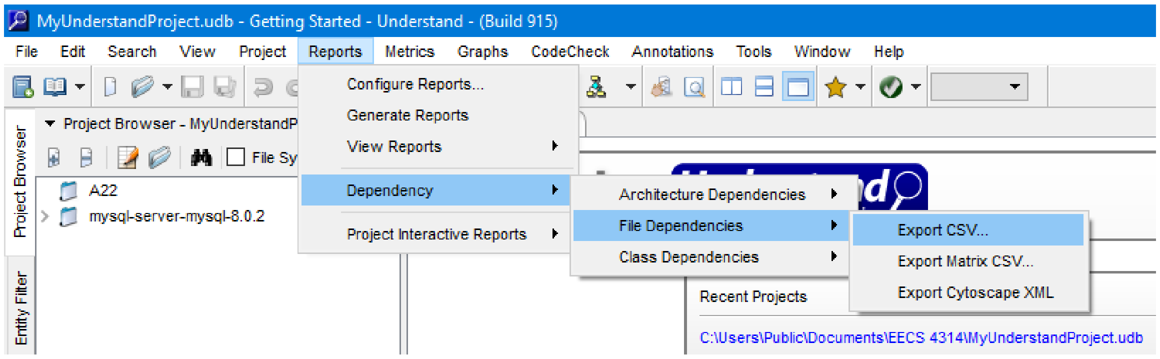
● Begin analyzing project sources. During the analysis you may be prompted for optional configuration items. They may add further information to your parsed results but are not required and can be safely ignored.

Configure more settings

○ Present the Project Configuration Dialog to view language specific configurations. All required configurations have been initialized to most common defaults. This can be invoked later at any time from the the Project menu

☑ Use the new project wizard when creating new projects

Step 6: Export Dependencies in a CSV File


MyUnderstandProject.udb - Getting Started - Understand - (Build 915)
File Edit Search View Project Reports Metrics Graphs CodeCheck Annotations Tools Window Help
Configure Reports...
Generate Reports
View Reports
Dependency
Project Interactive Reports
Architecture Dependencies
File Dependencies
Class Dependencies
Export CSV...
Export Matrix CSV...
Export Cytoscape XML
Project Browser - MyUnderstandP
A22
mysql-server-mysql-8.0.2
Recent Projects
C:\Users\Public\Documents\EECS 4314\MyUnderstandProject.udb

12.2 Appendix B: Explained Code

Appendix B-1: Dependency Extraction Script

```
import os
import sys

file_ta = open(sys.argv[2], "w")

for root, dirs, files in os.walk(sys.argv[1], topdown=False):
    for name in files:
        if name[-3:] == ".cc" or name[-2:] == ".h" \
                or name[-2:] == ".c" or name[-4:] == ".cpp" \
                or name[-4:] == ".hpp" or name[-5:] == ".java":

            if name[-5:] == ".java":
                find_include = "import"
                include_size = 6

            else:
                find_include = "#include"
                include_size = 8

            lines = open(os.path.join(root, name), "r")
            for line in lines:
                if line[:include_size] == find_include:
                    left_dependency = os.path.join(root,
name).replace('\\', '/')[os.path.join(root, name)
                            .find("mysql-server-mysql-8.0.2"):]
                    if name[-5:] == ".java":
                        line_copy = line
                        split = 0
                        pos = 0
                        while line_copy.find('.') != -1:
                            pos = line_copy.find('.')
                            split += pos + 1
                            line_copy = line_copy[pos + 1:]

                        string = "{} ->
{}.java\n".format(left_dependency, line[split:-2])
                    else:
                        string = "{} -> {}\n".format(left_dependency,
line[10:-2])

                    if string.rfind('"') != -1:
                        string = string[0:string.rfind('"')] + "\n"
                    if (string.rfind('>') != -1) & (string.count('>') >
1):
                        string = string[0:string.rfind('>')] + "\n"
                    if string[string.find('>')+2:-1].find('/') > -1:
                        string = "{} -> {}\n"\
                            .format(left_dependency,
os.path.basename(os.path.normpath(string[string.find('>')+2:-1])))
                    file_ta.write(string)
file_ta.close()
```

Appendix B-2: Initial Common Dependencies Script

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Scanner;

public class InitialMutualDependnecy {

      public static void main(String[] args) throws
FileNotFoundException, UnsupportedEncodingException
      {
            String file1 = args[0];
            String file2 = args[1];
            String output_file = args[2];

            Scanner file1_scanner = new Scanner(new File(file1));
            PrintWriter writer = new PrintWriter(output_file, "UTF-8");

            while(file1_scanner.hasNextLine())
            {
                  String line_f1 = file1_scanner.nextLine();
                  if(!line_f1.isEmpty())
                  {
                        Scanner file2_scanner = new Scanner(new
File(file2));
                        boolean found = false;
                        while(file2_scanner.hasNextLine() && !found)
                        {
                              String line_f2 =
file2_scanner.nextLine();
                              if(line_f1.equals(line_f2))
                              {
                                    writer.println(line_f1);
                                    found = true;
                              }
                        }
                        file2_scanner.close();
                  }
            }

      }

}
```

Appendix B-3: Final Common Dependencies Script

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Scanner;

public class MutualDependnecy {

      public static void main(String[] args) throws
FileNotFoundException, UnsupportedEncodingException
      {
            HashSet<String> deps = new HashSet<String>();
            HashSet<String> common = new HashSet<String>();

            String file1 = args[0];
            String file2 = args[1];
            String output_file = args[2];

            PrintWriter writer = new PrintWriter(output_file, "UTF-8");
            Scanner file1_scanner = new Scanner(new File(file1));

            while(file1_scanner.hasNextLine())
            {
                  String line = file1_scanner.nextLine();
                  if(!line.isEmpty())
                        deps.add(line);
            }

            file1_scanner.close();

            Scanner file2_scanner = new Scanner(new File(file2));

            while(file2_scanner.hasNext())
            {
                  String line = file2_scanner.nextLine();
                  if(!line.isEmpty())
                  {
                        if(deps.contains(line))
                              common.add(line);
                  }
            }
            file2_scanner.close();

            Iterator<String> it = common.iterator();
            while(it.hasNext())
            {
                  writer.println(it.next());
            }

            writer.close();


      }

}
```