Dependency Extraction

IntelliJ IDEA

EECS 4314
Dr. Jack Jiang
November 22th, 2019

Report Prepared by Lamport's Legion:

Paul Sison
Connor A
Aya Abu Allan
Jonas Laya
Jeremy Winkler
Damanveer Bharaj
Connor D

Table of Contents

1.0 Abstract

The goal of this report is to analyze and compare the outputs of three different architecture extraction software when executed with the IntelliJ IDEA IDE's source code. The three tools used for this procedure were "Sci Tools^{TM} Understand", "srcML", and a team-developed tool. The report consists of an overview of each extraction tool and the derivation process used to create an architecture extraction output that was comparable between each tool. Towards the team's expectations, each tool provided data with notable differences amongst each other. This data was then refined into both high level and low level quantitative and qualitative portions, which then provided more insight into why the differences were present. Quantitatively, it was found that the srcML tool found more total dependencies and unique "depended-on" files when compared to the other tools. A statistical analysis of the outputs then showed that our custom tool performed the same as srcML (precision: 96.5%, recall: 83.5%), and that Understand had slightly poorer results in part due to limitations of our comparison techniques. The results of this report bring to light the importance of using multiple tools to analyze software dependencies, as each methodology has its own advantages and disadvantages producing diverging data.

2.0 Introduction & Overview

This report provides a breakdown and analysis of the various findings when running different architectural extraction software on the source code for the IntelliJ IDEA IDE. The three methods of extraction utilized for this study are "Sci Tools^{TM} Understand", "srcML", and a team-developed tool. The first step of the process was to learn how to extract source code using each of the tools. Given that the extraction from each tool would be compared to one another, the output would also have to be comprehended in a format that allowed clean comparisons.

This report begins by providing a summary of each of the extraction methods. That is an overview of what each tool is, and the various functionality behind them. Then a detailed approach on how the IntelliJ IDEA IDE source code is extracted and analyzed using each tool is specified. After the completion of each extraction, the next step is to analyze the output for file dependencies and perform a comparison between each method. The output of each tool is run through some basic scripts created by the team in order to convert the extractions into a common interface that allowed comparisons between each other. The comparison between each output was done through both a quantitative and qualitative analysis.

A quantitative analysis of the extracted outputs consisted of high level and low-level metrics generated by running the outputs through various team-generated code. The first metric would be a count for the total number of dependencies generated per extraction tool. The number of unique "depended-on" files were then assessed to see how many unique files were being depended on over the entire system. Each method would then provide their own quantities which were compared for similarities and differences.

The next step of the report is statistical analysis. This portion consisted of analysis on specific file dependency instances per tool, in order to determine the tool's precision based on common truth grounds. An important technique used here is "Precision and Recall". This method consists of determining a technique's precision, defined as the number of correctly retrieved instances overall retrieved instances, followed by "recall", defined as the number of correctly retrieved instances over the total number of correct instances present [5]. "Precision and Recall" was used on the outputs of the three extraction tools and compared with one another to provide better insight on the integrity of each extraction method.

The report concludes with a discussion on the various areas of the study which were noteworthy to be difficult or simply peaked interests. A description of what we took away from these findings and lessons we learned about performing such tasks again was noted.

3.0 Architecture Extraction Methods (Derivation Process)

3.1 Understand

The first tool used for architectural extraction is a program called Understand. Developed by SciTools^{TM}, Understand is described as a powerful IDE with a key feature that differentiates itself from other IDEs. That is its static code analysis [6]. Essentially, Understand is capable of taking imported source code from programs like IntelliJ IDEA, and generate metrics which help developers understand the underlying elements used to produce the software. Understand portrays these statistics using various visuals such as graphs, tables and standard outputted files. This helps with poorly documented software since users can take advantage of Understands metrics to better understand the decisions and implementation designs developers used when producing their software.

For this study, Understand was used to generate file dependency .csv reports on IntelliJ IDEA's source code. The first step of this process was to create a new project in Understand and import the IntelliJ IDEA's source code. Understand then ran a full analysis on the source code allowing the tool to process all the code for the metrics required later. After this step, creating the required File Dependency .csv report was relatively simple. Using the toolbar to navigate, the following was traversed Reports->Dependencies->File Dependency->Export to CSV. Given that the static code analysis was performed during project creation and the analyzed source code did not change, later on, the file dependency report was produced relatively fast. This report was then run through some basic Java code in order to convert the document into the .ta format, which would be comparable with the outputs of the other tools.

Additionally, when studying Understand for further functionality, some key features were noted. Other than producing these dependency reports, Understand produced some basic and more advanced metrics. Some of the basic metrics were statistics such as the number of classes, lines of code and comment to code ratios. Some more complex metrics included the number of inactive code lines and the number of declarative and executable statements.

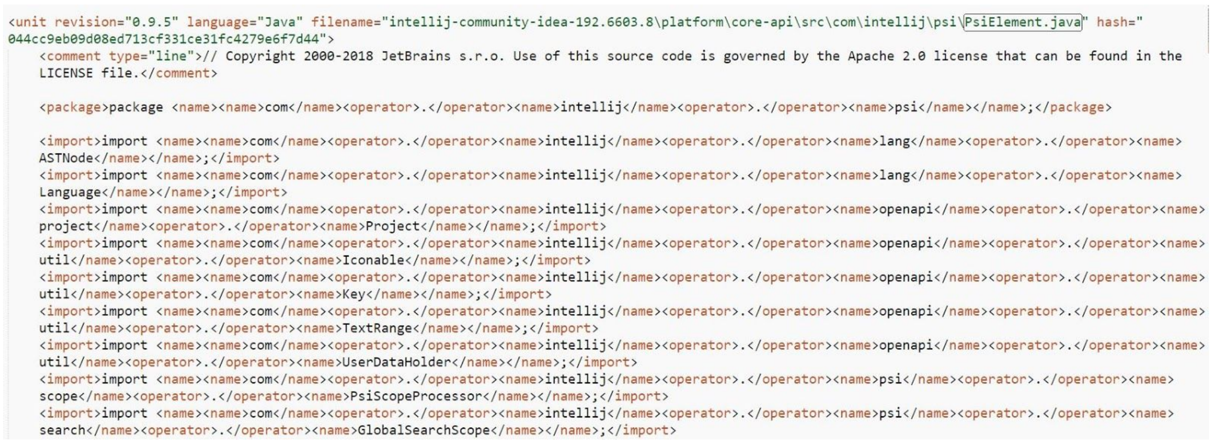
3.2 SrcML

SrcML is a command-line tool used to analyze, explore, and manipulate source code, which is made possible by leveraging XML to tag and keep track of all parts of your code, from function names to comments, and even whitespaces. It converts source code files into srcML format at a speed of 25 KLOC/sec and currently supports C, C++, C#, and Java, perfect for analyzing huge software systems like IntelliJ IDEA.

We started off the dependency extraction process by having srcML convert the entire IntelliJ codebase into XML. This was done by installing srcML on our system and executing the following command on a command-line prompt.

```
srcml intellij-community-idea-192.6603.8 -o intellij_idea.xml
```

The first argument is the root folder of the project source and the -o option is used to specify the name of the file where the resulting output will be stored. This produced a file that is almost 1 GB and nearly twice the size of the original code. The sample content shown below is a brief XML version of the source code file PsiElement.java along with its associated imports.



```
<unit revision="0.9.5" language="Java" filename="intellij-community-idea-192.6603.8\platform\core-api\src\com\intellij\psi\PsiElement.java" hash="
044cc9eb09d08ed713cf331ce31fc4279e6f7d44">
    <comment type="line">// Copyright 2000-2018 JetBrains s.r.o. Use of this source code is governed by the Apache 2.0 license that can be found in the
    LICENSE file.</comment>

    <package>package <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>psi</name></name>;</package>

    <import>import <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>lang</name><operator>.</operator><name>
    ASTNode</name></name>;</import>
    <import>import <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>lang</name><operator>.</operator><name>
    Language</name></name>;</import>
    <import>import <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>openapi</name><operator>.</operator><name>
    project</name><operator>.</operator><name>Project</name></name>;</import>
    <import>import <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>openapi</name><operator>.</operator><name>
    util</name><operator>.</operator><name>Iconable</name></name>;</import>
    <import>import <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>openapi</name><operator>.</operator><name>
    util</name><operator>.</operator><name>Key</name></name>;</import>
    <import>import <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>openapi</name><operator>.</operator><name>
    util</name><operator>.</operator><name>TextRange</name></name>;</import>
    <import>import <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>openapi</name><operator>.</operator><name>
    util</name><operator>.</operator><name>UserDataHolder</name></name>;</import>
    <import>import <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>psi</name><operator>.</operator><name>
    scope</name><operator>.</operator><name>PsiScopeProcessor</name></name>;</import>
    <import>import <name><name>com</name><operator>.</operator><name>intellij</name><operator>.</operator><name>psi</name><operator>.</operator><name>
    search</name><operator>.</operator><name>GlobalSearchScope</name></name>;</import>
```

Figure 1: PsiElement.java in srcML format

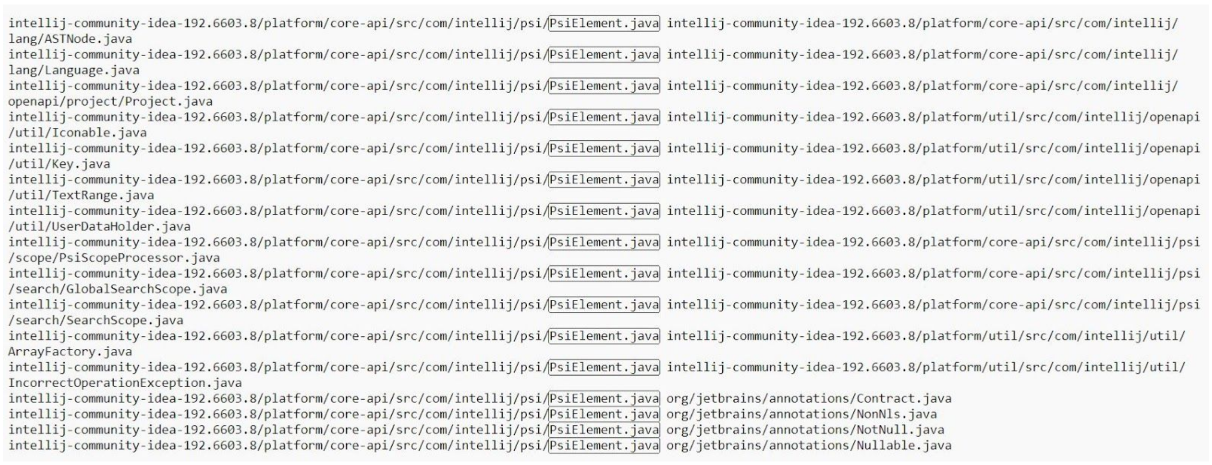
The srcML format allows the whole IntelliJ IDEA codebase to be queried using the -xpath option. For this assignment, the import statements made in Java source code files are extracted as they form a clear basis for a dependency. Other parts of the source code, such as the data types of the variables and functions declared in them can also be queried to determine if any other dependencies outside of the import statements are being made, but these are skipped for simplicity and time limitations.

The import dependencies are extracted using the following commands:

```
srcml --xpath "//src:import/src:name" intellij_idea.xml > intellij_idea_imports.xml
srcml --xpath "string(//src:unit/@filename)" intellij_idea_imports.xml > importing_class.txt
```

```
srcml --xpath "string(//src:unit/node())" intellij_idea_imports.xml > imported_class.txt
```

Some further processing was required to transform the results of the XPath queries into dependency pairings that are comparable to the results of the other two extraction methods. The `sed` command was used for some basic text manipulation. A Find/Replace tool that supports regular expressions can also do the job equally well. The grep command was used to filter out some lines that do not provide meaningful data, such as imports that take a whole package, *i.e*. ones that end with the * wildcard. The `paste` command was used to combine the *importing_class* and *imported_class* sections to form the dependency pairing. Lastly, a java program was used to complete the location path (within IntelliJ) of the imported classes.



```
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/
lang/ASTNode.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/
lang/Language.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/
openapi/project/Project.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/util/src/com/intellij/openapi
/util/Iconable.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/util/src/com/intellij/openapi
/util/Key.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/util/src/com/intellij/openapi
/util/TextRange.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/util/src/com/intellij/openapi
/util/UserDataHolder.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi
/scope/PsiScopeProcessor.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi
/search/GlobalSearchScope.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi
/search/SearchScope.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/util/src/com/intellij/util/
ArrayFactory.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java intellij-community-idea-192.6603.8/platform/util/src/com/intellij/util/
IncorrectOperationException.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java org/jetbrains/annotations/Contract.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java org/jetbrains/annotations/NonNls.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java org/jetbrains/annotations/NotNull.java
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java org/jetbrains/annotations/Nullable.java
```

Figure 2: Well-formatted dependency pairings

The whole extraction process using srcML is best summarized by the following flow diagram.

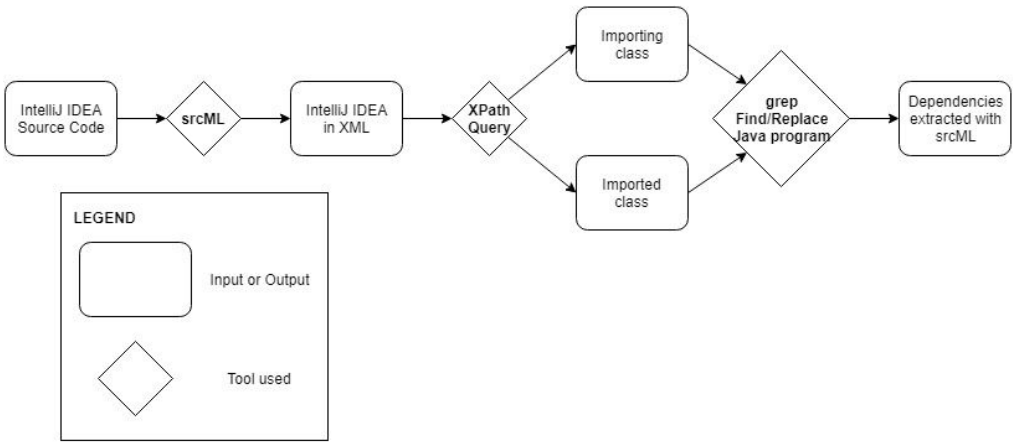


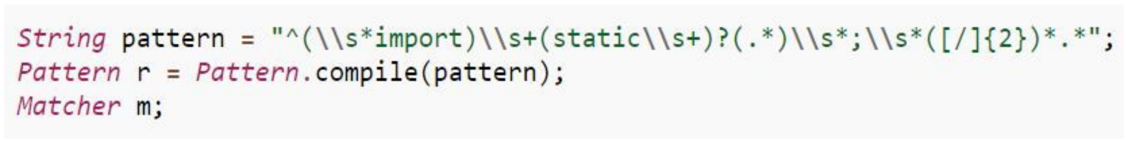
Figure 3: srcML extraction process

3.3 Custom Java program

The third tool we used for extracting dependencies is a Java program. The program scans all Java source code files in the IntelliJ IDEA codebase for its import statements to form a dependency between the source code file and the classes it imports.

The first step in this extraction process was to obtain a list of all the Java source code files in IntelliJ IDEA along with its complete path from the root folder. A separate Java program created this listing and stored the results in a file.

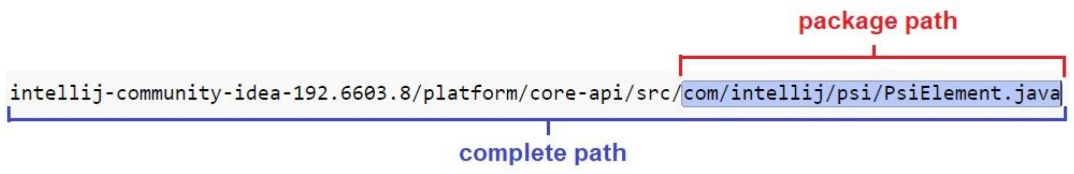
Our extraction tool then scanned each Java source code file on the list for import statements in its first 100 lines. Regular expressions were used to match lines with imports and to extract the class being imported.



```java
String pattern = "^(\\s*import)\\s+(static\\s+)?(.*)\\s*;\\s*([/]{2})*.*";
Pattern r = Pattern.compile(pattern);
Matcher m;
```

Figure 4: Regular expressions used in the Java program

Using the same Java file listing created earlier, key-value pairings were made between the package path of a class and its complete path from the project source root folder. These key-value pairings were then stored in a HashMap so complete paths of the imported classes can be easily accessed. Storing the dependencies with complete paths are important to ensure uniqueness and for comparability with the output of the other extraction methods



```
intellij-community-idea-192.6603.8/platform/core-api/src/com/intellij/psi/PsiElement.java
```

.

Figure 5: Package to Complete path of an imported class

The following is a sequence diagram of the extraction process using our custom Java program.

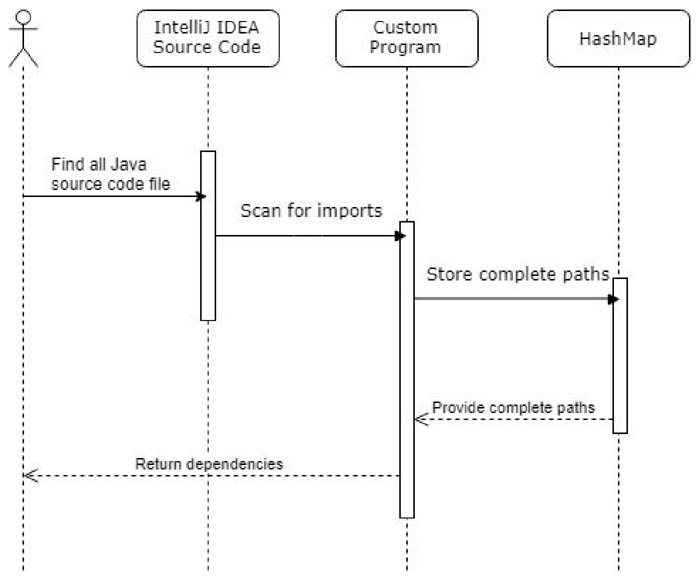
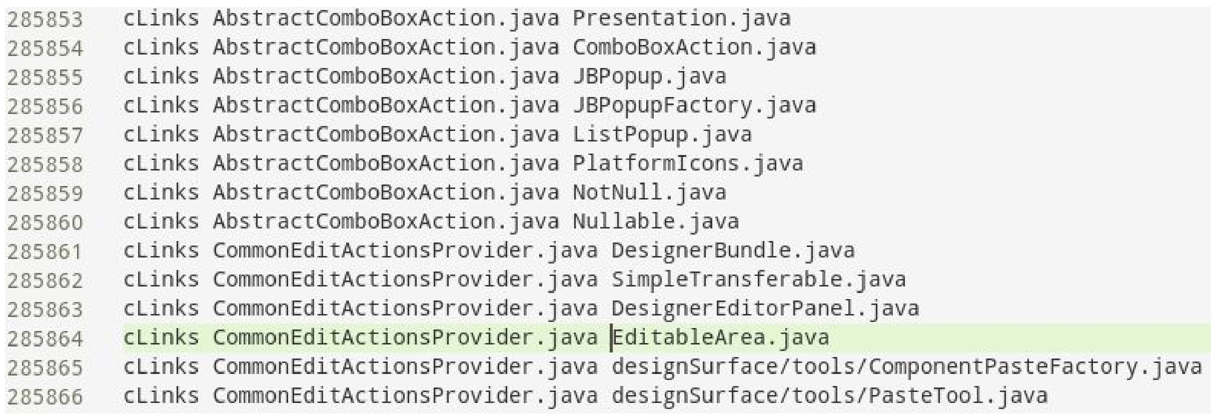


Figure 6: Extraction process using Custom program

4.0 Architecture Extraction Comparison

4.1 Dependency Collection Derivation & Analysis

The first step of the extraction comparison was getting output from our extraction methods that could be quantitatively compared. This was done by exporting the results of each extraction method to a .txt file with raw data from the corresponding extraction method. Once this raw data is created, some simple scripts were written to convert the raw extraction data to usable .ta data that can be analyzed.



```
cLinks AbstractComboBoxAction.java Presentation.java
cLinks AbstractComboBoxAction.java ComboBoxAction.java
cLinks AbstractComboBoxAction.java JBPopup.java
cLinks AbstractComboBoxAction.java JBPopupFactory.java
cLinks AbstractComboBoxAction.java ListPopup.java
cLinks AbstractComboBoxAction.java PlatformIcons.java
cLinks AbstractComboBoxAction.java NotNull.java
cLinks AbstractComboBoxAction.java Nullable.java
cLinks CommonEditActionsProvider.java DesignerBundle.java
cLinks CommonEditActionsProvider.java SimpleTransferable.java
cLinks CommonEditActionsProvider.java DesignerEditorPanel.java
cLinks CommonEditActionsProvider.java EditableArea.java
cLinks CommonEditActionsProvider.java designSurface/tools/ComponentPasteFactory.java
cLinks CommonEditActionsProvider.java designSurface/tools/PasteTool.java
```

Figure 7: Example of our dependency extraction tool's .ta output file. Pathnames have been simplified for this image.

Once the dependencies were laid out in a workable format, we began developing an extraction tool to collect some basic metrics from the different file outputs. The initial model of our tool was able to determine the total amount of dependencies in the system provided by a specific extraction method. As expected, for each of the three tools, we found different results for the total amount of dependencies. Our results can be found in **Figures 8 and 9**.

Dependency Extraction Method	Number of Total Dependencies
Understand	370,140
SrcML	475,712
Our Tool	329,899

Table 1: Table of total dependencies found by each extraction method

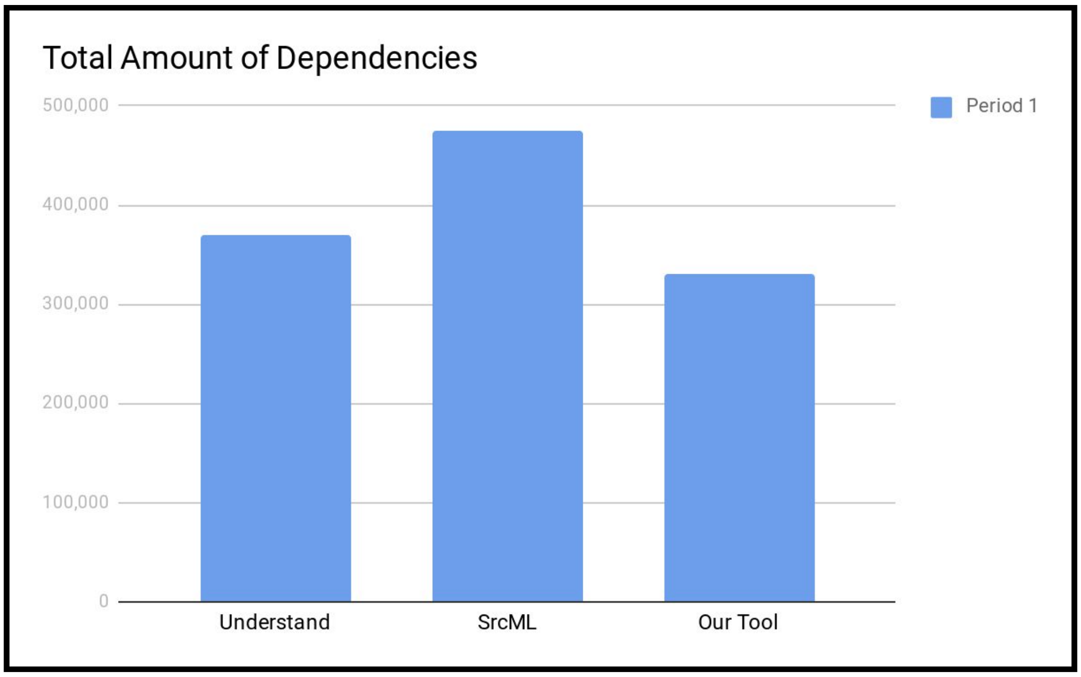
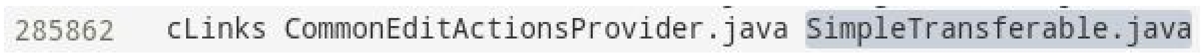


Figure 8: Graph of data in **Table 1**

At this (very high) level of analysis, we can see that there are significant differences in the different extraction methods. However, without looking deeper, there isn't much more insight this data can provide us just yet.

The next step was analyzing the structure of the data. This was to see what meaningful data could be found from a line of our output. In **Figure 9**, you can see a line for yourself. The structure goes (*relationship-file1-file2)*. The cLinks relationship denotes a dependency from file 1 to file 2.



```
285862   cLinks CommonEditActionsProvider.java SimpleTransferable.java
```

Figure 9: Example of a dependency, with the Aggregated or "Depended on" file highlighted

In a small system, we typically look at files that have the most dependencies on other files. This is because there's a significant risk of these files being "Superman" classes, holding too much functionality that should be shared amongst the system. However, with IntelliJ, it's more important to look at the classes that are being aggregated/composed. In the example above, this is the second file listed on a line of data. These "depended on" files are more important when we look at IntelliJ because of the sheer amount of dependencies in the overall system. If we look at files with a lot of dependencies, we'll find data types, utility files, singletons and other development structures that are integral to the IntelliJ platform.

After realizing the importance, the amount of "Depended on" files were now tabulated in a similar sense to the total dependencies. You can find this data in **Table 2** and **Figure 10**.

Dependency Extraction Method	**Number of Total Dependencies**	**Number of Files Depended On**
Understand	370,140	25,419
SrcML	475,712	30,557
Our Tool	329,899	19,411

Table 2: Table of data collected thus far

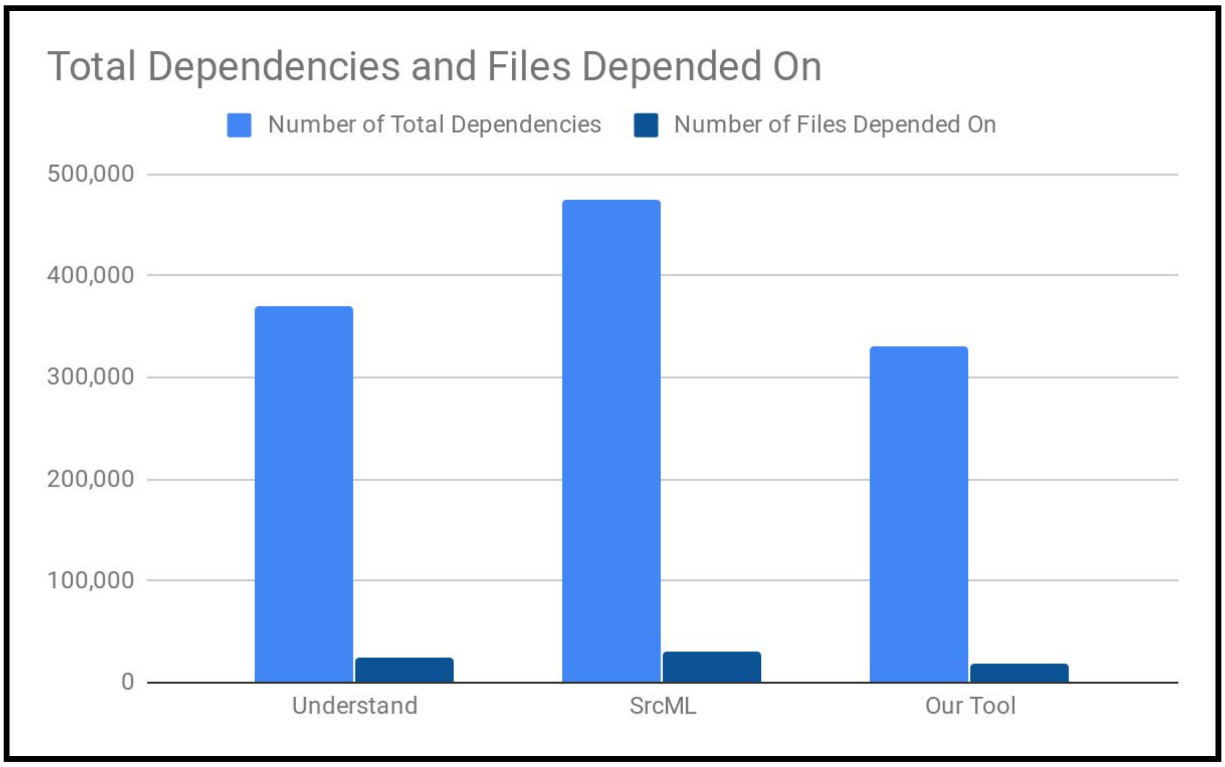


Figure 10: Graph of data in **Table 2**

Seeing the files depended on data alongside the total dependencies in the system beside each other make their relationship look nearly linear, which doesn't signal any obvious differences between the different extraction methods. This meant we had to look deeper.

To get any valuable insights on the system, we needed to look at the relationships individual files actually had. The tricky part of looking for relationships like this is the sheer amount of data that needed to be sifted through. Luckily, in our previous analysis, we found that files that were depended on by a lot of files are important components in the overall system. This conclusion leads to another. The files that have the most "depended on" relationships are some of the most crucial parts of the system. Since these have large significance, we looked at the top 9 depended on files, and looked at how many files depended on those top 9 for comparison.

Understand		SrcML		Our Tool	
Filename	**Depend on Count**	**Filename**	**Depend on Count**	**Filename**	**Depend on Count**
NotNull.java	24,664	NotNull.java	24,446	NotNull.java	22,128
Nullable.java	13,784	Nullable.java	13,575	Nullable.java	12,428
Project.java	8,800	List.java	10,033	Project.java	8,271
PsiElement.java	8,696	Project.java	8,728	List.java	6,852
VirtualFile.java	4,449	PsiElement.java	4,504	PsiElement.java	4,217
PsiFile.java	3,942	ArrayList.java	4,332	VirtualFile.java	3,577
StringUtil.jav	3,920	VirtualFile.java	4,216	StringUtil.java	3,469
Logger.java	3,584	StringUtil.java	3,847	Logger.java	3,284

Table 3: Top 8 Depended on files with # of dependencies on each

Our tool accounts for more dependencies than the top 8, but to keep figures at a reasonable size, we've held it down to just the top 8. We feel that this accurately represents the majority of the system, and any important insights made about the system can be seen from this top 8.

If we look at file names, we can see a similar ordering amongst the different extraction methods. Generally, a file found in one list can be found within +/- 4 positions on another dependency list, usually a smaller difference. Where we see significant differences between the different methods is some of the counts. **PsiElement.java** has an error range of almost 4000 files dependant on it. We can also see that SrcML picked up some Java base libraries (**List.java, ArrayList.java**). Also generally the differences of files depending on these files aren't linear with the other relationships graphed above.

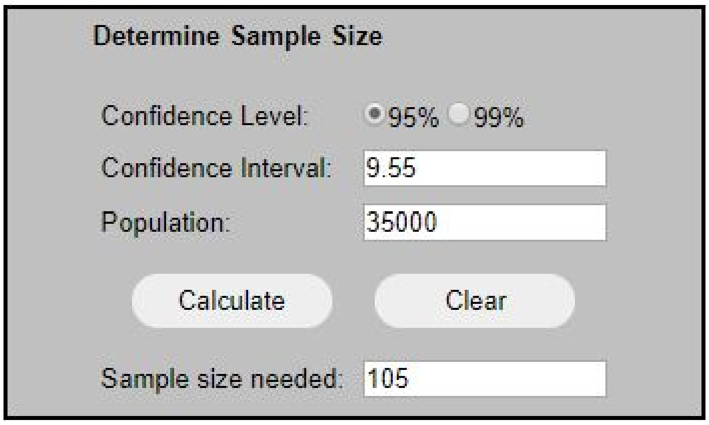
These findings lead us to realize that the differences between the extraction methods are completely systematic. They all suffice as viable extraction methods, but some different conclusions of the usage statistics on data types and utility classes can be made depending on the extraction tool your project depends on.

4.2 Statistical Analysis

A precision and recall statistical analysis were performed on the three different extraction techniques in order to complete the quantitative analysis of the system.

Methodology

For our comparison, we decided to take a sample size of the total number of files instead of all dependencies. This way, we were able to reduce the sample size while still revealing the same results. In order to be able to compare the three techniques, the first step was to take a random sampling of files to use in our analysis. With a total population of approximately 35000 files, we chose a sample size of 105 files in order to have a maximum 9.55% confidence interval.


Determine Sample Size

Confidence Level: [✓ 95%] 99%

Confidence Interval: 9.55

Population: 35000

Sample size needed: 105

Figure 11. SurveySystem Sample Size Calculator Results [7]

The sampling was performed by using a pseudo-random number generator and then randomly selecting 105 filenames from the master list of all files in the IntelliJ repository.

Once the sampling was complete, each team member was tasked with finding the ground truth values for 15 of those files. The results of this ground-truthing were collected, cleaned, and organized into a processing document for comparing against the dependency lists of the other methods. The cleaning was necessary because of the possibility of some of the pathnames becoming incongruent across the different extraction techniques; our methodology compares only class-name dependency matching and does not look for full path/package names in the comparison.

After cleaning, the techniques were ready for comparison. Using the newly created processing file, we created a program that would scan through and build a ground truth list of dependencies for each of the samples, and then cross-reference that list with the tuple-attribute files that were created from each of the dependency extraction techniques. An example of a sample analysis is shown below:

```
Now Analyzing CustomTagDescriptorBase.java:

  Now comparing against Custom Analysis:
    CustomTagDescriptorBase.java -> PsiMetaData.java dependency found.
    CustomTagDescriptorBase.java -> PsiWritableMetaData.java dependency found.
    Manual /\ Custom: PsiMetaData.java PsiWritableMetaData.java

    Number of correct custom dependencies was: 2
    Number of all custom dependencies was: 2
    Precision of Custom = 1.0000
    Recall of Custom = 1.0000

  Now comparing against Understand Analysis:
    CustomTagDescriptorBase.java -> JspElementDescriptor.java dependency found.
    CustomTagDescriptorBase.java -> PsiMetaData.java dependency found.
    CustomTagDescriptorBase.java -> PsiWritableMetaData.java dependency found.
    Manual /\ Understand: PsiMetaData.java PsiWritableMetaData.java

    Number of correct understand dependencies was: 2
    Number of all understand dependencies was: 3
    Precision of Understand = 0.6667
    Recall of Understand = 1.0000

  Now comparing against srcML Analysis:
    CustomTagDescriptorBase.java -> PsiMetaData.java dependency found.
    CustomTagDescriptorBase.java -> PsiWritableMetaData.java dependency found.
    Manual /\ srcML: PsiMetaData.java PsiWritableMetaData.java

    Number of correct srcml dependencies was: 2
    Number of all srcml dependencies was: 2
    Precision of srcML = 1.0000
    Recall of srcML = 1.0000
```

Results

The precision and recall was gathered across all 105 samples and the results can be found in the table below:

	Understand	**srcML**	**Our Tool**
Precision	65.6%	96.5%	96.5%
Recall	79.8%	83.5%	83.5%

Table 4. Precision and recall for sampled files across each tool.

4.3 Qualitative Analysis

Our tool ended up aligning exactly with srcML, which makes sense because they both only look at the import statements and reject anything with “.*”’s. In some cases, the ground-truthing was able to find the “.*” dependency, and these are the cases where those tools are falling short in the recall. The precision may be due to other errors in the programs or also perhaps errors on the parts of our team members in the manual identification of the true dependencies.

The most surprising result was that of Understand, coming in with only a 65.6% precision. The recall is demonstrably close to the other techniques (and well within the confidence interval), but the precision is significantly worse than the others. Upon investigating some key examples (like the one shown above), we can get a better idea of what's going on. In the example above, Understand found all the files that the manual extraction did (hence, 100% recall in this example), but also found an additional dependency of JspElementDescriptor.java which the others did not. This was an inherited class for the file, which notably our other methods do not account for when it is in the same package as the file in question. This is the kind of issue that causes the large gap in precision between Understand and our other methods - the fact that Understand is capturing inheritance whereas the others are not.

5.0 Risks and Limitations

As mentioned previously, srcML and our tool did not consider things such as * dependencies or inheritance. Understand did consider many of these intricate cases which our tool and srcML did not. This is likely a large reason why when considering the precision and recall of the different extraction techniques, that Understand did so poorly in the precision analysis.

Another limitation was the total number of dependencies totalling over 1 million. With this large number of dependencies, we noticed that some of our scripts we created were taking long to run, which is understandable with the size of the data we were considering.

6.0 Lessons Learned

As discussed throughout the report, none of the extraction methods had the exact same results. They were mostly similar in their results to the point that conclusions made from each method would be fairly similar, if not the same.

At first, it seemed like Understand was the most accurate since it picked up the most dependencies, but then we don't see Understand picking up base libraries like SrcML. With all the differences added together, we don't get all the information, but we have an idea where each tool is lacking. Understand does, on the other hand, have a way of tracking inherited classes that were not covered by the other techniques.

This highlighted for us the importance of using multiple tools and sources for information based on different codebases and projects. The collection of information from different sources - similar to watching the news or references for an English assignment - gives us a better picture of the overall situation that we're trying to understand and work on.

The statistical analysis proved to be much more complicated than originally thought, and our team has gained a lot of appreciation for manual data analysis and ground-truthing. We're very thankful for the larger team size to also take advantage of the pipelining of this work. Additionally, this was an excellent exercise in creating shell scripts to filter through and sort files for comparison.

7.0 Conclusion

In conclusion, dependency extraction and analysis is not a single-tracked process. It's clear that there are different tools with different capabilities and limitations, but not necessarily clear which is better. The determining factor for which is better will be found by asking what insight is trying to be gained by the analysis. Ultimately, all of the extraction methods can be used to get a rough idea of the software architecture, but Understand and srcML do provide some valuable (and very different) capabilities that future analysts wish to utilize.

8.0 References

1. JetBrains, "IntelliJ Platform SDK DevGuide," *JetBrains IntelliJ Platform SDK*, 12-Mar-2019. [Online]. Available: https://www.jetbrains.org/intellij/sdk/docs/welcome.html

2. IntelliJ IDEA Architecture and Performance, GoogleTechTalks, Dmitry Jemerov, 2008, Youtube Video

3. PSI Navigation Demo, IntelliJ-SDK-docs, JohnHake, JetBrains, https://github.com/JetBrains/intellij-sdk-docs/blob/master/code_samples/psi_demo/src/com/intellij/tutorials/psi/PsiNavigationDemoAction.java, Accessed 2019

4. Lamport's Legion, "Conceptual Architecture IntelliJ IDEA", Lamport's Legion EECS 4314, 07-Oct-2019. [Online]. Available: https://jeremywinkler.github.io/pdfs/conceptual-arch.pdf

5. Wikipedia Contributor, "Precision and Recall", Wikipedia, 18-Nov-2019. [Online]. Available: https://en.wikipedia.org/wiki/Precision_and_recall

6. Sci Tools, "Understand Legacy Code Tool". Sci ToolsTM Understand, [Online]. Available: https://scitools.com/legacy-code-tool/

7. Survey System, "Sample Size Calculator". Creative Research Systems, 19-Nov-2019 [Online] Available: https://www.surveysystem.com/sscalc.html