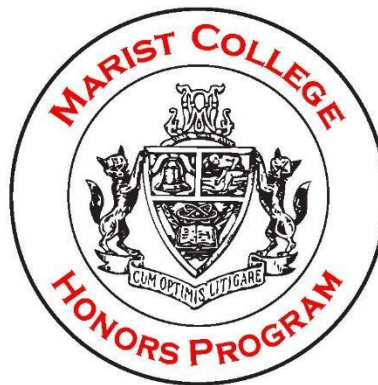


# SOURCE CODE MODELING BASED ON USER INTERACTION: A JAVA PARSING APPROACH

Ryan Andrew Sheffler

Prof. Michael Gildein, Faculty Advisor

Spring 2021



Submitted to the Honors Council in partial  
fulfillment of the requirements for the degree of  
Honors in Liberal Arts

## **Table of Contents**

|                                  |           |
|----------------------------------|-----------|
| <b>Introduction</b>              | <b>3</b>  |
| Understanding Parsing            | 3         |
| A Different Approach             | 6         |
| Potential Real-World Application | 7         |
| <b>The Script</b>                | <b>7</b>  |
| The Concept                      | 7         |
| Development Environment          | 10        |
| The Parser                       | 11        |
| The Modeler                      | 14        |
| <b>Results</b>                   | <b>17</b> |
| <b>Conclusion</b>                | <b>25</b> |
| <b>Appendix</b>                  | <b>26</b> |
| Works Cited                      | 26        |
| Online Exhibit                   | 27        |
| Resources Used                   | 28        |

## **Abstract**

There is an old adage, "A picture is worth a thousand words." Modeling and visualizing the intricacies and complex relationships in systems such as software follows this principle. In a previous Honors By Contract project, I created a program to parse Java programs and produce an analysis of how the individual parts interact with each other and the user. The goal of this project is to extend that work to generate a three dimensional, or 3D, model of the digested data the parser produces. Other similar code parsers typically focus on simply improving code. What can be exploited? How can my code be more efficient? What may cause an error as libraries and other programs are updated in the future? These are all very important questions to be asked, and the answers can be quite valuable for developers. However, they are not the only questions to ask. The solution I propose takes that analysis of how the program interacts with itself and the user and presents it visually to reduce the difficulty for new or returning developers to understand how exactly large projects work and comprehend the dependencies among the parts. The model can show how changing one small part can affect other segments of the program.

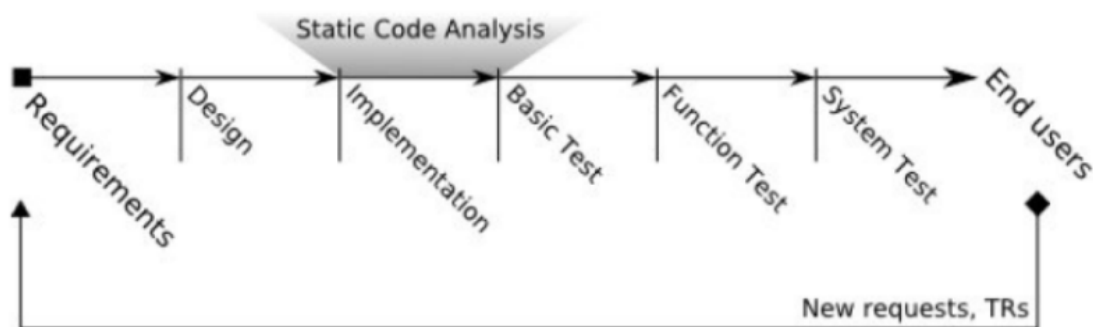
## **I. Introduction**

### **i. Understanding Parsing**

It isn't enough to simply write code that works. It's important to write code that works well and continues to work well for a long time. As time goes on, new technologies will be developed and old technologies will change. Developers need to keep this in mind. What is the environment I'm creating code to run in? Who am I creating code for? What works best in this language, and what syntax is prone to creating errors? How might these things change, and what precautions need to be taken to make sure my creation won't break when changes are inevitably made, that it won't topple in the slightest breeze? However, just as with anything humans create,

coding issues can often be difficult for us to properly identify. Code parsers work to provide an excellent set of objective eyes in these scenarios. Code parsers are programs designed to read and process code, with the intention of providing insights to that code. Some level of parser is built into most integrated development environments (IDEs), working in real time. As the programmer types, the parser will highlight obvious code issues such as potential crash-causing errors, inefficiencies such as unclosed Scanner objects in Java, and unnecessary or unused methods and file imports in real time. Though quite reliable by this point, these parsers are simple, mostly just sparing programmers the time it takes to compile their code by quickly highlighting errors beforehand, almost as a spellchecker does for text.

More popular, though, are standalone code parsers. After writing code, especially large projects, many programmers will choose to put their code through a parser such as PMD or SonarSource for review [1, 2]. In his paper on the subject, Alexandru G. Bardas uses this figure (Figure 1), which describes these parsers' place in the development cycle well.



*Fig. 1 [10]*

These parsers are much less concerned about quick results and are more focused on comprehensive ones. Parsers like these will provide all sorts of recommendations and warnings, ranging from what simpler parsers would identify to pointing out what may soon no longer be supported and suggesting a different way of doing that task that isn't at risk of deprecation. When

one mentions code parsers, these parsers are typically being discussed, though the more accurate term would be “static code analyzer.” As Panagiotis Louridas puts it in his paper on the subject, “No machine can substitute for good sense, a solid knowledge of fundamentals, clear thinking, and discipline, but bug detection tools can help developers,” which is precisely what these parsers are: Another tool in a programmer’s arsenal to ensure the best code. He also notes that many parsers look for current errors that go beyond asking if it will compile or not. To summarize, he shares a simple Java program that, while compiling and running with no issues whatsoever, will not operate as the programmer desires. Running it through FindBugs, which is an open source static code checker for Java developed at the University of Maryland, returns a few possible larger scale issues, all of which are relevant within the context of the program [12]. These errors, though not picked up during compilation or by the IDE’s built-in parser, not only cause the program to not work as intended, but could cause fatal errors that were never properly accounted for when certain conditions are met [11].

Static code analyzers can go a bit beyond just errors, too. As Louridas says, “people tend to fall into the same traps repeatedly.” While this is very useful for pure errors and bugs, this can also be applied to vulnerabilities. Whether they’re used unknowingly by a programmer or intentionally by a malicious attacker, the security vulnerabilities of a program are something programmers are always looking to minimize. People constantly make mistakes, and programmers will log known mistakes into databases. When analyzing just the text of code, the typical parser has a database of known weaknesses that usually includes common security pitfalls. Some analyzers go a bit further, stress testing programs by attempting to predict all possible values of variables and how they may trigger runtime errors or cause the program to act

in an unintended way [11]. Through this, static code analyzers become not just a way to generically improve code, but to secure it as well.

## **ii. A Different Approach**

These parsers provide extremely useful insights and will always be very valuable to programmers everywhere. However, the questions they ask and answers they provide aren't the only ones. When I examined the approach other parsers take, I found a few questions of my own to ask. Most parsers take a comparative look at the code they're given. "Based on other code, how can this code's efficiency be improved? Comparing this code to what else I have access to, does anything appear to be directly copied? I have access to an encyclopedia of various errors, does anything here appear to potentially cause one?" What if a parser were instead to take a more introspective look? The idea for this code parser was instead to focus on two points of interaction: interaction between the program and the user and interaction between the program and other parts of itself. The parser would attempt to break down these points into forms that are easier to process. The intra-program interactions are removed from their context — simply letting the user know where the program is calling what other files — and the user interactions are summed up as a simple score. This parser is different in that it is not attempting to inherently provide advice to improve the project on its own, but rather simply insights into the code's connections.

A second part of this project came along with that thought. If the parser broke down the points of interest into simpler forms, why not have a simpler way to view it? The points of interest that the parser collects could be fed into a modeler to produce an image of the data. A diagram or model would present the desired information in a much more easily digestible way. Depending on the amount of information, a three dimensional (3D) model may be an interesting

implementation of this diagram. Allowing the user to manipulate a model in a 3D space would allow them to understand the information presented in a much more intimate way.

### **iii. Potential Real-World Application**

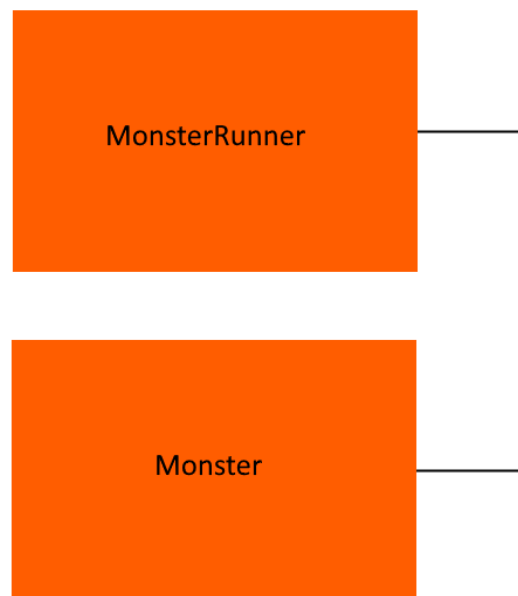
Other parsers have an obvious use case in code improvement. Upon getting to a working or “finished” state, a programmer would use a parser to help find and fix bugs before they get encountered and to generally improve the code before release. However, the typical outsider notion of a single programmer working tirelessly on a piece of code from start to finish is a rare occurrence at best. Most projects in the technology industry are created and supported by a team of designers and programmers, designers and programmers who are constantly being shifted around among various projects. Despite this, they’re still expected to know the ins and outs of any given project or to learn them rather quickly. If they were asked to quickly become familiar with a large project they had never seen before, which would be more useful at a glance between the project’s detailed, intricate documentation and a simple picture diagramming how the program connects with itself? This is the primary use-case of this project, to aid in the comprehension and understanding of larger projects.

## **II. The Script**

### **i. The Concept**

The project was quickly divided into two parts, the parser and the modeler. I made the decision to hold the two as separate, independent programs. The parser component could be given a program, figure out the important information, then pack it away in a file. From there, the modeler component could pick up the pre-digested data the parser created for it, and present it visually. From here, two things had yet to be decided. Firstly, what information would the parser look for and record? Secondly, how would this data be modeled in an intuitive way?

These were decided with a bit of a backwards approach. It was instead asked, “What would be good to represent in a model?” As a result, two quick sketches of a possible final look were created based on an old and simple program created for an elementary Java course. Here, *MonsterRunner.java* is the main method, being what the user would run to begin the program, which calls on *Monster.java* to create an object of the class *Monster* and store data within it. The first sketch was a simplified view:

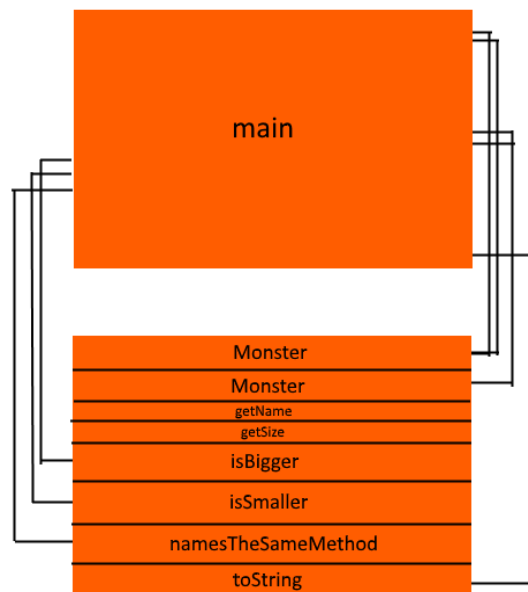


*Fig. 2*

With the view presented in Figure 2, the programmer can visualize the connections between files. With a small and trivial program like this one, there isn’t much to see, but for a large program with a file count reaching the triple digits, this view alone could be helpful in understanding the scope and dependencies of the program. Although it’s hard to see because the two files under analysis are almost identical in line count, the size of the blocks themselves are scaled based on the class sizes. Not pictured was an intended 3D element where files that interact more with the user would be placed in front of those that interact less. In this case, only



*MonsterRunner.java* writes to or reads from the console, the only interaction this program uses, so it would be placed more towards the screen by default, with the programmer being able to rotate the model to see this depth more clearly. However, this level of detail is often not enough when visualizing all the connections in a program, resulting in this concept sketch of a more detailed view:



*Fig. 3*

Here, the methods of the files are exposed and individual connections are shown. This level of detail much more effectively communicates the exact dependencies of files in relation to each other. By not showing the actual code, the amount of information the parser has to pack into a file is reduced and the file is still simplified somewhat. It is common practice (and courtesy) to name methods based on their function anyway, so just including the names would likely allow a new pair of eyes to learn a great deal about the class without having to be concerned with the minutiae of how exactly the method accomplishes its goal. More details were discussed as well, such as tracking what arguments a method would take or what variables existed within a class or

are passed through method calls, but it was decided that while that information may be useful, it could also result in an information overload that makes the diagram much harder to understand and could be difficult to code with more complex programs.

From these idealized concepts, the information required to create them can be extrapolated. Obviously, the filename should be stored, alongside an index of which file it is in the list to have a backup identifier and primary key if the unusual event of multiple identically-named files existing occurs. The size of the file could be reduced to a simple integer by recording its line count. To assemble the connection lines, three pieces of data need to be put together: the call out location, the destination file, and the destination line number. In practice, the lower detail of only one line between files with a connection can be extrapolated from the higher detail list so, in the interest of keeping the result file smaller, no additional list is needed for that. A list of methods must be assembled with their starting positions to form the detailed view block divisions. Methods in most programming languages (like Java) have a specific structure or even keyword used to define them, making this possible to detect. Lastly, the files would need to be scored based on their interactions with the user to determine where along that axis they should lie. For this, it was decided to score the files. As the files are parsed through, certain strings could be searched for. Each time they're encountered, an amount can be added to a score for that file, which would then be included in the results file. With that, both prerequisite questions were now fully answered and the actual programming of the solution was ready to begin.

## **ii. Development Environment**

This project was created entirely in Java, simply due to familiarity with the language. Also, because of convenience and familiarity, Eclipse was chosen as the IDE. Because outside

libraries were to be used to simplify parsing and 3D graphic generation, Apache's Maven was integrated to ensure these dependencies would be installed and used automatically [9]. Three of these outside libraries were used, which will be discussed as they become relevant in explaining the function of the two components. Again, due to familiarity, the parser was kept to digesting Java.

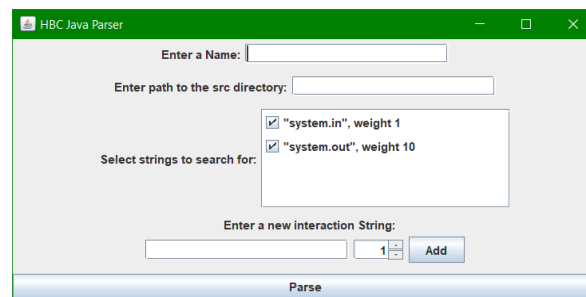
### **iii. The Parser**

The parser component was created first, ensuring that the information the modeler would eventually be receiving was predetermined. The basic plan for the function of the parser was to take in a file directory and results file name, confirm the directory exists, collect all the Java files in that directory, run through each while paying attention for the information outlined in Section II.i, package each into a JSON object, group each together into one large JSON object, then write that object from a variable into a permanent file using the collected results file name. The JSON, or JavaScript Object Notation, filetype was chosen for its simple readability being a plain text file of keys and data, and the fact that this project didn't require any sort of security or encryption. Using the JSON filetype also allowed the file to easily be written and later read by the program. More information on the exact notation and syntax of this filetype can be found at the website [JSON.org](http://JSON.org) [3]. Unfortunately, while JSON support is built-in with Javascript, it is not built-in to Java. To solve this, the JSON In Java library was imported through Maven. Out of the many different available libraries designed to bring JSON support to Java, this library was chosen because it was created and is maintained by the actual creators of the JSON filetype.

Next was the issue of actually parsing files. In the simplest scenario, the Java files might be read as plain text and iterated through that way. However, in search of another, better solution, parser modules were evaluated. Eventually, the simply yet aptly named `JavaParser` was

integrated into the project. JavaParser allowed for the easy collection of the text from the files it was pointed to, as well as a few extra tools to aid in the collection of the desired information. With both of these outside libraries imported, the program was ready to function.

The first matter to implement was collecting information from the programmer, specifically a file path and a name for the output file. In the first builds of the program, this was simply done through the console using a basic Java Scanner and, while this functionality is still intact, a GUI, or Graphical User Interface, was later added, pictured here:



*Fig. 4*

While discussing the GUI, the additional functionality it brought to the project should briefly be discussed. To put it simply, what constitutes an interaction with the user? Originally, this was hard-coded to have only the two default options present in the list in Figure 3, “System.in” and “System.out.” With the GUI and a few modifications to the code, it became possible for the programmer to add search strings. As the final result is an integer “score,” each string is also weighted with a number that is added to the score whenever the string is encountered.

Moving back to the beginning of the process, the parser has collected the target file path and a name. From here, the program needs to get the files to scan. It does this by scanning the given directory and all of its subdirectories and creating a Java File variable for each. Then, the parser narrows down to just the Java files by creating a separate Array and adding to it only the

Files whose name ends in “.java”. Now, all the starter information is collected and the parser is ready to, well, parse.

The parser creates a JSONArray object that will later be serialized to a string and written to the output file. For each of the input Java files, a parse method is run to collect the desired information from the file as a JSONObject, which is added to the aforementioned JSONArray. Within this parse method, a few bits of information are collected prior to a main loop iterating through the code. Firstly, the file’s name is recorded, alongside an array position that is passed into the method to serve as an identifier. Thanks to the tools that JavaParser offers, the file is easily split into an array of strings based on line, and the file’s line count is gathered from the length of this array. JavaParser also offers a method to collect the names of all methods in the target file, which is a quick and easy way to get that list. Unfortunately, this does not also collect any information about where the method starts or ends, so that proposed visual representation was rejected. The parse method now enters a loop where it looks at each line of the code given to it and searches for the relevant information.

The first and perhaps most dangerous pitfall the parser keeps an eye out for are comments. Comments could contain anything, including the names of other files in the system or even the defined interaction phrases, which could trick the parser into believing a file is interacting with another or the user. As a result, comments are the first thing looked for. If a comment is found, it will be completely trimmed from the line or lines it affects. Then, the line will be reread. Next, the parser looks for the mention of any other filename. If it finds one, it records the filename and the current line it is on. Unfortunately, it would be hard to implement a destination line without parsing multiple files at once. Although this could be done, the difficulty of this led to it being put on the backburner and ultimately scrapped in favor of other features.

The last thing the main loop looks for is a match with one of the user interaction strings. If one is found, then its weight is added to the file's interaction score. A list of interaction lines is also kept in the final file, although it ultimately was unused in the final model.

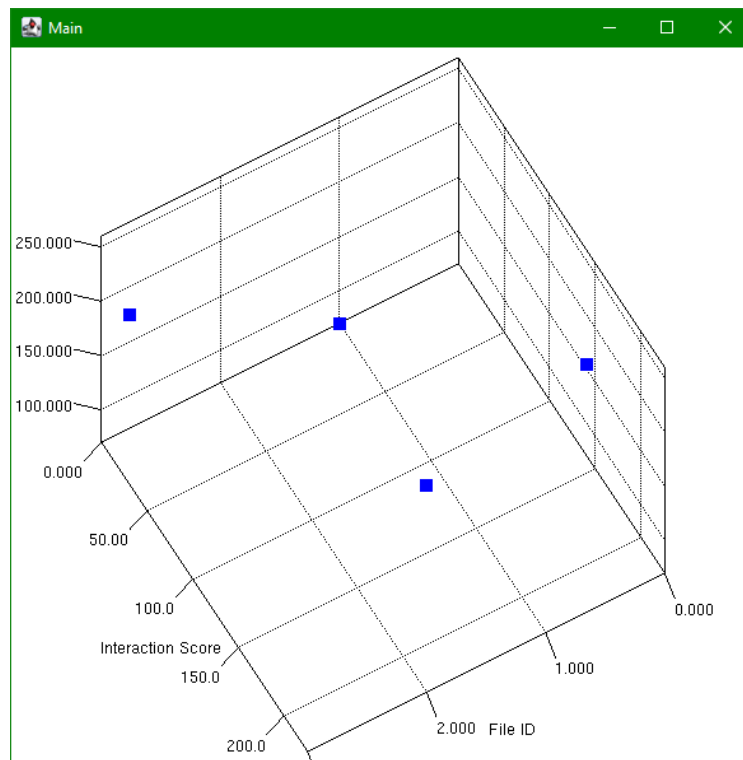
From here, the parser is in its end phase. Because JavaParser offers access to this through a convenient method, it was chosen to add a list of the file's imported files to the stored information. It packages all of those in JSONObjects, then in the previously set up JSONArray. It then uses the "stringify" function to turn that encoded object into readable text. It then assigns that to a File object and creates a file of the ".json" filetype with the name the user gave back at the beginning of this process. From here, the parser has completed its function and exits.

#### **iv. The Modeler**

After the programmer has a digest file from the parser, it's time to use the modeler. The file need only be placed in the program's working directory (typically `git\HBC-Java-Parser\hbcModel\target\classes\marist\thesisModeler`), then the name given to the program when prompted on startup.

The hope for the modeler component was to create a 3D model. While one was made, time constraints as well as the dedication to using only Java for this project did force some concessions. After looking at a few different libraries, most of which would require additional non-Java programming, such as Python scripting, or for the programmer to have outside programs like Gephi, the Jzy3D library was chosen. Unfortunately, this library does not work on Apple OSX computers at all. Linux has not been tested. Also of note is that, while this library works and creates the desired graph with no issues on Microsoft Windows, it does create several warning messages upon startup. These are present even in the developer-created demos, and don't appear to have any effect on the program. While it uses only Java, it is more limited to only

set graph types. As a result, the graph is broken into two main parts, a 3D scatter plot displaying each file as a point, and individual file's detailed views. The 3D view for a small, four file project appears as follows:



*Fig. 5*

Each point is a file. The three axes they're placed on are the Interaction Score, File ID, and Linecount. Larger files will be higher, files that interact with the user more will be more towards the bottom right, and the last axis helps separate the points. This graph is fully manipulatable, so the programmer can spin, scale, and zoom to get the desired view. To differentiate these files and get to the detailed view, another window is opened:



Fig. 6

This window shows a list of the files in the project. Moving the mouse cursor over a button will highlight the point on the graph in a different color. Clicking on one of the buttons will open yet another window, displaying details of that file.

Here is MainRunner's detail window. Displayed here is a vertical rectangle whose size in pixels is equal to double of the file's line count. Each callout is represented by a colored line. These alternate sides, alongside the size being double the line count, makes sure that no lines will overlap. Listed below the file's graphical representation is a panel of three tabs. Each tab is self-explanatory: The Legend tab displays each other file it calls to in the color that it is represented by, the Methods tab lists each method within the file, and the File Imports tab shows each of the file's import statements. There is one big issue with



Fig. 7

this implementation. Despite significant efforts to the contrary, the window as a whole will not scroll. Resizing the window hides the lower portion of the window, rather than properly bring up a scrollbar. The main problem with this is that large files will simply run off the end of the screen with no way to see the bottom. On a standard 1920x1080 monitor, the limit is around 400 lines.



### III. Results

With the parser and modeler completed, all that remains is to test them against some files to see what it produces. To run a first test, let's take a look at the program the original concept sketches were based on. Also in the directory are a few other files, from the same topic in the elementary Java course that the *Monster* and *MonsterRunner* classes were constructed for. Despite their presence, they do not interact with *Monster* or *MonsterRunner* and will be ignored here. The 3D graph of these files can be seen in Figure 8, with *Monster* being the largest (highest along the left axis) and *MonsterRunner* being highlighted:

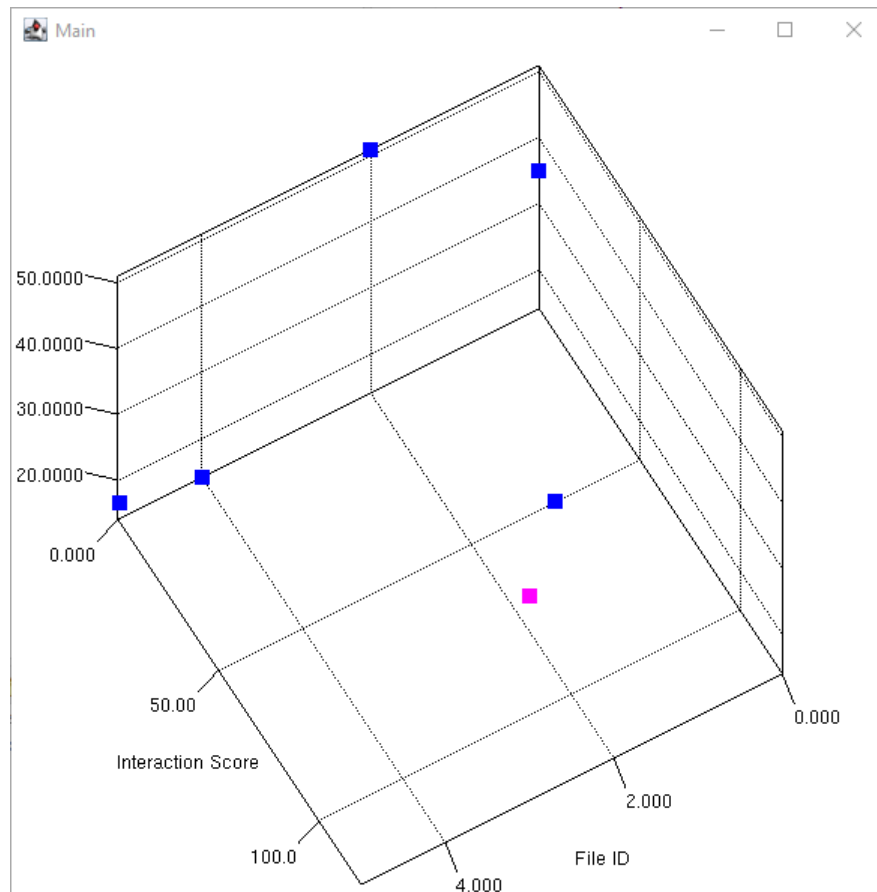


Fig. 8

Already from this graph, something that was noted but unable to be displayed in the original sketches can be seen, which is that *MonsterRunner* interacts with the user much more. Also, by manipulating the graph to get a better look at the Linecount axis, it can be noted that the two files are nearly identical in size, something which is further backed up by their detail windows:

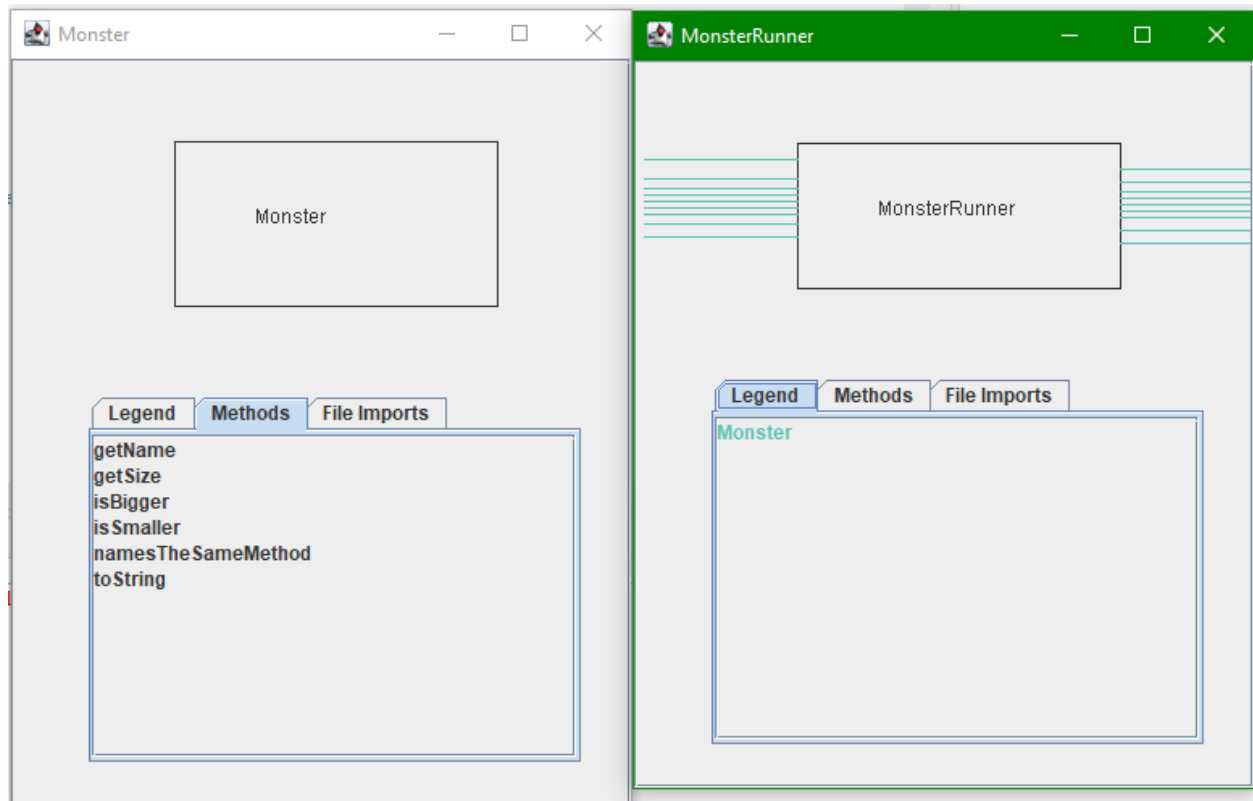
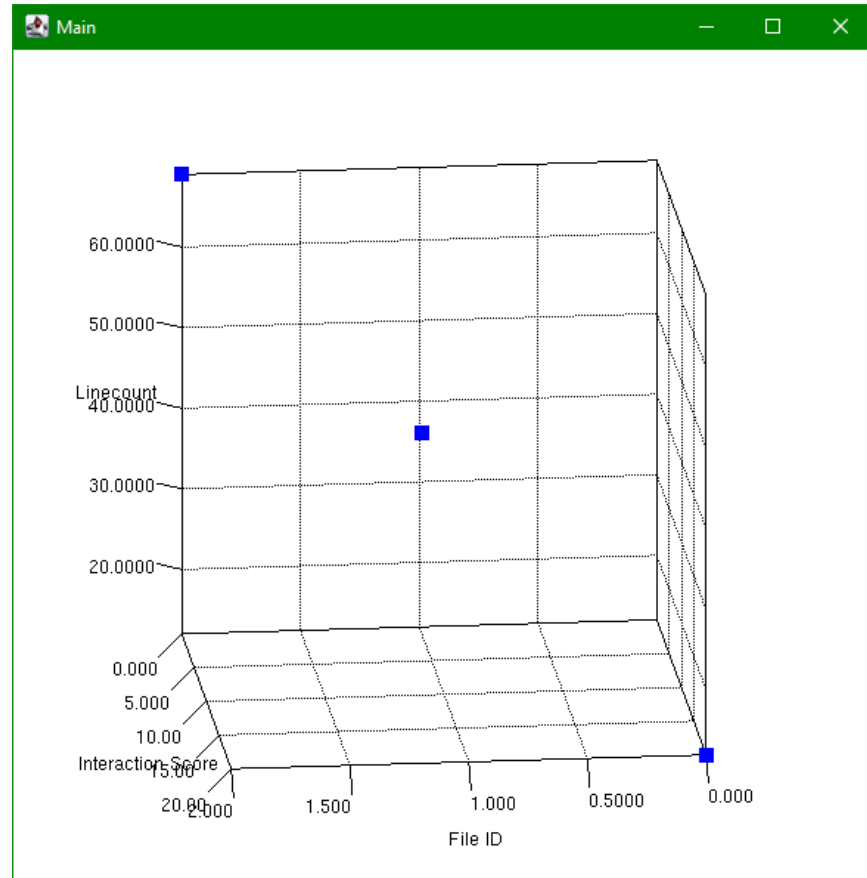


Fig. 9

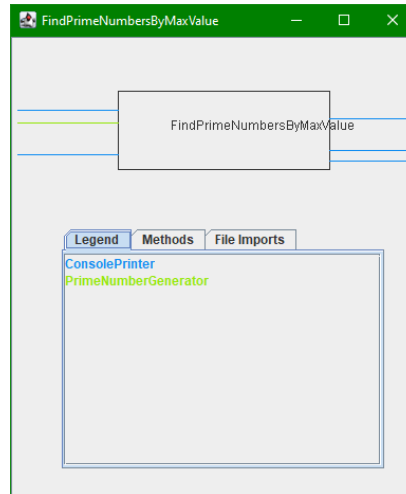
The windows are nearly identical in size, which means the classes are the same as well. Shown again for us to see is every callout *MonsterRunner* makes, positioned referentially at its corresponding line. Since *Monster* never makes any calls, the Methods tab is more interesting to look at, giving a quick look at everything the class can do. But this is something we've already seen, so let's take a look at some other projects. Luckily, SonarCloud offers hosting of open-source code. Projects that opt in can provide a link to their GitHub repository for anyone to

easily view or download their code. Grabbing some Java projects from this website can let us see how the parser and modeler handle projects of different sizes and what larger projects, like the intended use case, might look like with this final view. Let's start small, with a SonarCloud project simply called "demo" [4]. According to the associated GitHub repository, it is a simple prime number generator. Its 3D graph looks like this:



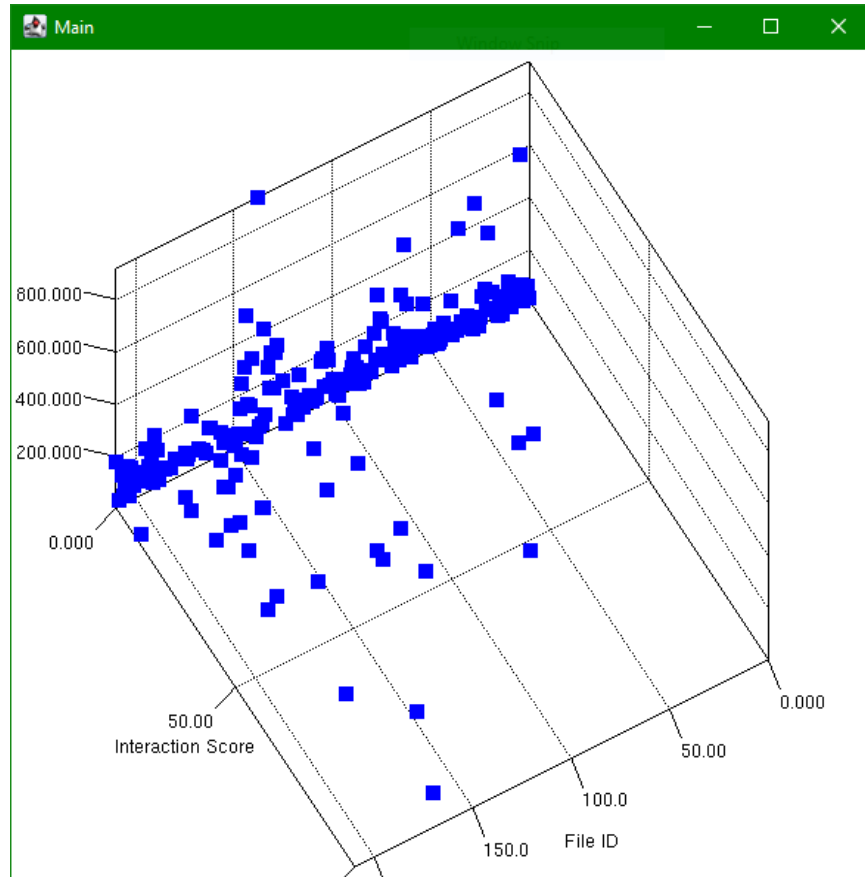
*Fig. 10*

Right away, it's clear that only one file handles interactions with the user through the console, which is the conveniently named `ConsolePrinter`. Inspecting the detail windows, only one is of particular interest, which is `FindPrimeNumbersByMaxValue`.



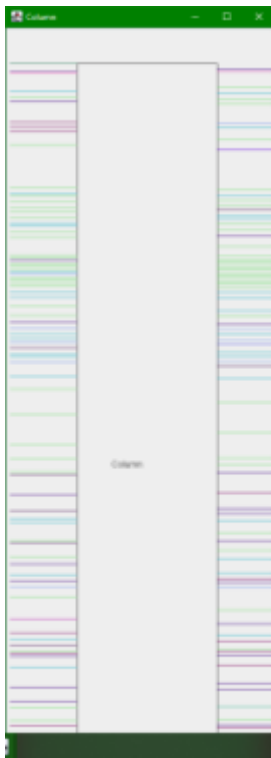
*Fig. 11*

This file is the only one that calls out to others to use their utilities, and it contains the main method. The other two files are clearly there just to provide support methods to this one. Let's move on to something slightly larger, from 136 lines of code to around 15,000. This project was called “molgenis-emx2” on SonarCloud which, after some research, appears to be an open source bioinformatics data management system [5, 6].



*Fig. 12*

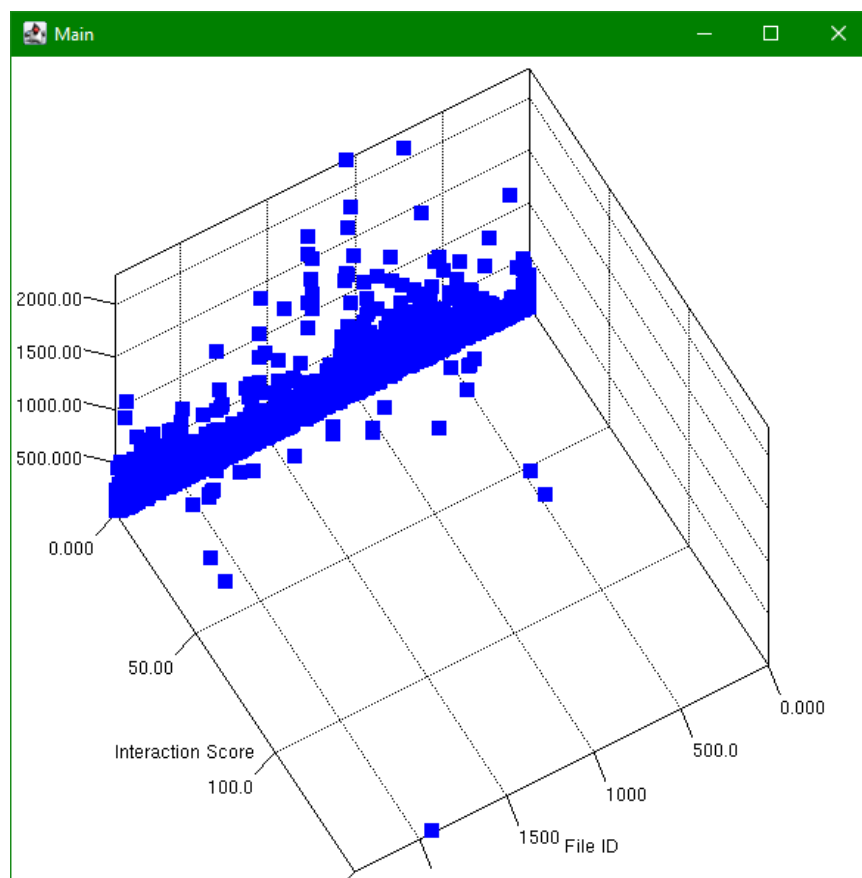
Already, one thing is clear: This graph is a lot more cluttered than what we've seen before. However, the big contrast between this blue and the bright pink that was selected as the highlight color makes it clear which file is which, even in the big cluster of files. That choice was a decent one. However, examining a few of these brought to light an error that was briefly addressed at the end of Section II.iv, which is illustrated well in the Column file pictured in Figure 13.



*Fig. 13*

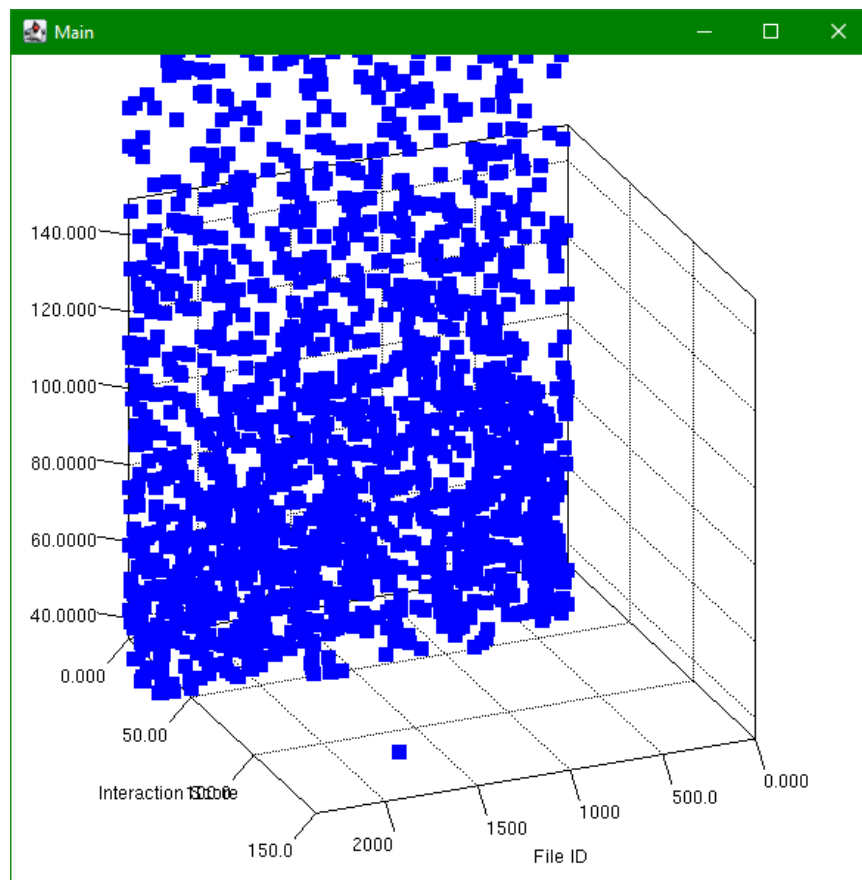
In the case of a window being too large, the contents of the window were placed in a scroll pane. However, due to a strange error whose exact roots are unknown, this scroll pane refuses to work as it is

intended to. Resizing the window or forcing any kind of window size will not display any scroll bar, but instead will simply erase lower sections of the window. It is possible to circumvent this problem as it does rely on screen size. A screen with a larger vertical resolution will allow the programmer to see the lower regions of the window. The other files could be examined to learn more about this particular program, but in the interest of including a variety of different programs, let's take another jump in project size. The last project to look at is much larger, around 140,000 lines of code. This project is entitled "DDF Reactor" on SonarCloud, which is at least a part of Codice's Distributed Data Framework [7]. Based on the information on their website, DDF works as a sort of information aggregator, with their description of its function being "DDF is an interoperability platform that provides secure and scalable discovery and retrieval from a wide array of disparate sources. DDF is built to "unzip and run" while having the ability to scale to large enterprise systems" [8]. Right away, the difference such a large project makes is apparent. The parsing, which was nearly instant on a higher end desktop even on the Molgenis program, took somewhere between 1 to 2 minutes. And the graph?



*Fig. 12*

In areas, this is nearly solid blue. Oftentimes, highlighted files can be hard to see as points will be behind others, but being able to manipulate the graph freely makes it much easier to tell them apart. I haven't shown off the zooming or scaling functions as of yet, but this graph is an excellent one for that. By zooming and Z-shifting the graph downwards, one can “separate the pack,” making that solid blue a bit more decipherable:



*Fig. 13*

It wasn't addressed much, but one thing of interest in each 3D graph is the horizontal axis, the Interaction Score axis. Generally, there is one point or a small group of points that sit far beyond the others as outliers. As expected, programs generally have only one or two classes that handle user interactions. These exact results may be a little skewed, however, as larger projects may choose to have their own methods of interacting with end users. These files were only parsed with the default console calls in mind, as I was not familiar enough with them to know to include any custom methods. Still, being able to tell where what the user sees is coming from, or which files are "front-facing," can also be a valuable tool to a programmer. If they need to make a change to the UI, for example, they now know exactly where to look. Peering into a few of the detailed file windows, there isn't too much that isn't reminiscent of what we've already seen in



previous examinations. If one wanted to, the programmer could certainly scan through these windows with the graph as a guide to attempt to understand the program, either to understand it in the first place or gain a better understanding of it. In the interest of time, addressing more specific files will be skipped here, but it should be noted that the larger standard filesize highlights the detail window size issue and makes its refusal to work somewhat more frustrating.

## **IV. Conclusion**

Code parsing is a very important field for programmers, similar to how useful spellchecking and proofreading utilities are for writers. However, while one angle of examining code may be the most useful or common, that doesn't mean there aren't other useful bits of information to glean from that same code. While parsing code for the sake of improvement is very important for a programmer to do when they have finished coding, a new set of eyes may not understand what they are looking at at a glance. Examining the dependencies and interactions of code specifically can be great both for helping newer developers to digest the hundreds of lines of code they're looking at and for existing developers to understand the web of dependencies and the impact that editing one file may have on another, to predict errors such as incompatible arguments before they happen.

This project is far from perfect, mainly in the modeler component. The parser could use a few fixes and quality of life improvements such as using a file selection window instead of just a text input for the file path, but many concessions were made from the original model design in order to create something that worked while still being understandable. The ultimate goal would be to return to something much closer to the original concepts, mainly in creating one interconnected graph instead of several related ones. The current form of the modeler does list the connections between files, but it would be much better if it was able to visually show these

connections. To add to this, it may be worth it for total clarity purposes to take the extra time and effort to determine not only the origin point of inter-file calls, but their destination points as well. These improvements and more could be added in the future to further the goals of this project.

## V. Appendix

### i. Works Cited

- [1] PMD. 2012. pmd/pmd: An extensible multilanguage static code analyzer. (July 2012). Retrieved June 6, 2021 from <https://github.com/pmd/pmd>
- [2] SonarSource. 2021. Automatic Code Review, Testing, Inspection & Auditing. (2021). Retrieved June 6, 2021 from <https://sonarcloud.io/>
- [3] Anon. Introducing JSON. Retrieved June 19, 2021 from <https://www.json.org/json-en.html>
- [4] Onur Casun. 2021. demo - Onur Casun. (June 2021). Retrieved June 23, 2021 from [https://sonarcloud.io/dashboard?id=onurcasun\\_prime-number-generator](https://sonarcloud.io/dashboard?id=onurcasun_prime-number-generator)
- [5] Department of Genetics of the University Medical Center Groningen. 2021. molgenis-emx2 - molgenis. (January 2021). Retrieved June 24, 2021 from [https://sonarcloud.io/dashboard?id=molgenis\\_molgenis-emx2](https://sonarcloud.io/dashboard?id=molgenis_molgenis-emx2)
- [6] MOLGENIS Team. Retrieved June 24, 2021 from <https://www.molgenis.org/>
- [7] The Codice Foundation. 2017. DDF Reactor - codice. (September 2017). Retrieved June 24, 2021 from <https://sonarcloud.io/dashboard?id=ddf>
- [8] The Codice Foundation. DDF: Intelligent Integration. Retrieved June 24, 2021 from <http://codice.org/ddf/>
- [9] Brett Porter, Jason van Zyl, and Olivier Lamy. 2021. Welcome to Apache Maven. (2021). Retrieved June 27, 2021 from <https://maven.apache.org/>
- [10] Alexandru G. Bardas. 2010. STATIC CODE ANALYSIS. (2010). Retrieved June 27, 2021 from <https://core.ac.uk/download/pdf/6552448.pdf>
- [11] Panagiotis Louridas. 2006. Static Code Analysis. (2006). Retrieved June 27, 2021 from <https://www-proquest-com.marist.idm.oclc.org/docview/215840070?accountid=28549&amp:pq-origsite=primo>
- [12] Anon. 2015. Find Bugs in Java Programs. (March 2015). Retrieved June 27, 2021 from <http://findbugs.sourceforge.net>
- [13] Iván Ruiz-Rube, Tatiana Person, Juan Manuel Dodero, José Miguel Mota, and Javier Merchán Sánchez-Jara. 2019. Applying static code analysis for domain-specific languages. (April 2019). Retrieved June 27, 2021 from <https://link.springer.com/article/10.1007/s10270-019-00729-w>


ii. Online Exhibit

Ryan Sheffler

Source Code Modeling based on User Interaction: A Java Parsing Approach

**Ryan Sheffler**

Class of 2021, School of Computer Science



I'm Ryan Sheffler, a senior working on my Bachelor's in Computer Science all here at Marist. I grew up equipped with a GameBoy. I remember the fun I had as a little kid fooling about with the few games I had and how much I learned from them. Those experiences are where my fascination with computers began. From playing with computers, I developed an interest in learning how they work and how to make them operate. As time passed, my interests only grew and I began learning computer programming and eventually decided to pursue Computer Science through high school and as my degree in college.

**Abstract**

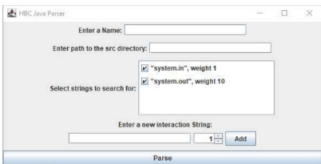
There is an old adage, "A picture is worth a thousand words." Modeling and visualizing the intricacies and complex relationships in systems such software follows this principle. In a previous Honors By Contract project, I created a program to parse Java programs and produce an analysis of how the individual parts interact with each other and the user. The goal of this project is to extend that work to generate a 3D model of the digested data the parser produces. Other similar code parsers typically focus on simply improving code. What can be exploited? How can my code be more efficient? What may cause an error as libraries and other programs are updated in the future? These are all very important questions to be asked, and the answers can be quite valuable for developers. However, they are not the only questions to ask. The solution I propose takes that analysis of how the program interacts with itself and the user and presents it visually to reduce the difficulty for new or returning developers to understand how exactly large projects work and comprehend the dependencies among the parts. The model can show how changing one small part can affect other segments of the program.

Code Parsing?

For those unfamiliar, *code parsing* is the process of taking in code and processing it. Specifically, it refers to a program doing that. A *code parser* is a program that exists to read other programs, digest them, and provide certain insights to them. A common example is *PMD*. Parsers are great for developers, allowing a new set of eyes to quickly be run over their code, reviewing it for any potential errors. Most common parsers will also offer code suggestions and insights, such as pointing out what may soon no longer be supported and suggesting a different way of doing that task that isn't at risk of deprecation. Simpler parsers are also built into most IDEs, highlighting and notifying developers of issues as they type. Standalone parsers tend to try to go deeper, providing the developers with all kinds of feedback on their code, from simple formatting to providing alternatives to inefficient code.

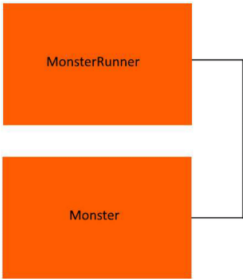
My Implementation

As discussed in the *Abstract*, my parser keeps its eyes open for a different set of things from most parsers. As it reads through each file in its target directory, it takes a note of the other resources it imports, the methods of the class, and where it mentions other files in the directory, among a few other things. Most importantly, though, it keeps track of where the file interacts with the user, and generates a score based on how much it interacts with the user. What counts as an interaction with the user can be defined by the user of the parser, but defaults to the standard "System.out" and "System.in" calls. It then stores these in a file, to be used by the modeler component of my project.



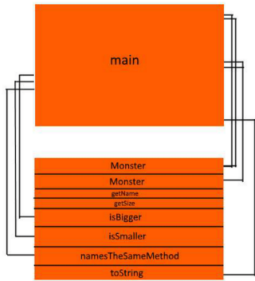
A view of the parser component's GUI.

The modeler unfortunately isn't yet complete. However, I do have a simple mockup to show. In this case, I took a simple 2 file/class program I wrote while learning Java. *MonsterRunner.java* is the main method, being what the user would run to begin the program, which calls on *Monster.java* to create and store data. A simple view may look something like the following:



A low-detail model of the program.

This simple view just shows the files and their direct connections. With this trivial program, this view isn't particularly interesting, but with a larger program, this alone could be valuable information. But where does the 3D aspect of this model apply? I couldn't demonstrate it here, but that is determined by the user interaction score the parser gave the file. In this case, *MonsterRunner* is the only one printing to or receiving information from the console, so it would be more "in front" or closer to the viewer than *Monster*. What the end user sees is a very important thing to developers, so it's important to know what is shaping their view. What if we wanted to see a bit more information, though?



A sample model showing more detail about the program.

My parser keeps track of what methods are in a file and where it calls other files in the directory. A higher detail view could reveal these methods, and show these more advanced connections, making it easier to see just what depends on what else. This unique relational view of a program is great for developers who are unfamiliar with a project. In the technology industry, developers are constantly being shifted around projects and a major difficulty with this is the speed at which they're able to adjust to their new project. Seeing a simplified model of a program represented visually like my modeler produces makes it much easier to wrap one's head around a large program quicker, meaning it takes much less time for a new developer to truly pick up and become a member of their new team.

My current code (and paper when it's completed) can be found at my [GitHub repository](#) [here](#).

- iii. GitHub Repository  
<https://github.com/RYGUY722/HBC-Java-Parser>

- iv. **Resources Used**

JSON In Java: <https://mvnrepository.com/artifact/org.json/json>

JavaParser: <http://javaparser.org>

Jzy3D: <http://www.jzy3d.org>