Department of Computing Science
Software Maintenance and Evolution

PINOT

Pattern INference and recOvery Tool

Authors:

Swastik Satyanarayan Nayak (S4151968)
Sytse Oegema (s3173267)

Lecturer:

Dr. Mohamed Soliman

Coach:

Filipe Capela

GitHub Repository:

https://github.com/Swastik-RUG/SoftwareEvolutionPinot

December 10, 2020

Contents

1 Introduction

Software systems build in the Object Oriented(OO) paradigm consist of multiple objects that deliver the total functionality of a system. In the design process of large OO systems design patterns can be used to increase coherency, maintainability and quality of the system. Design patterns define best practices for frequently occuring problems in the form of an abstract OO design. A system can easier be analysed when the applied design patterns indicate the relation between the different objects in a system. Design patterns come especially in handy when analysing undocumented legacy systems. Pinot(Pattern INference and recOvery Tool) is a pattern recognition tool that can be used to analyse the design patterns used in Java applications[1]. Our goal is to analyse the structure of PINOT and refactor it's code structure. In order to achieve this we will first perform literature research on the topic of reverse pattern engineering in section 2. Where particular interest is direct to PINOT and the reclassification scheme of GoF patterns that it uses for the purpose of reverse engineering[1, 2]. Then we will use reverse engineering tools to analyse the structure of PINOT in section 3.
We will start by understanding the background of this project where we discuss the Gang of Four (GoF patterns) and the Pinot application in Section 2. In Section 3 we will first look at how to prepare the Pinot application for analysis. We have analyse the application in two ways. First, the source code is translated to Abstract Syntax Tree (AST) using the Clang compiler, and compiled to cpa files. The cpa files are loaded to the Structure 101 C/C++ tool[1] for visualization. Second, the source code is first processed through the Scitools Understand [2] and a UDB file is generated, which is then loaded to the Structure 101 generic tool[3] for visualization. The section will also list the observations made in these tools with respect to the Pinot source code, followed by a discussion of the cohesive clusters and the architecture of the project. Finally we will conclude our results in section .

2 Background

Pattern reverse engineering can be achieved in roughly 2 different fashions. The first category of reverse engineering approaches identifies design patterns on their their structural aspect. This is achieved by identifying the relationships and inter-dependence between classes. These relationships can be extracted from class-inheritance, interface hierarchies, class methods and attributes, and return types. This extracted information is represented in language-independent structures, such as a Abstract Syntax Tree(AST) or Abstract Syntax Graph(ASG). Patterns can then be identified by mapping abstract design pattern structures against the language-independent structure of the system[3]. The structural design pattern recognition approach is graphically represented in figure 1.

[1] https://structure101.com/binaries/structure101-studio-cpa-win64-5.0.16419.exe
[2] https://www.scitools.com/
[3] https://structure101.com/downloads/

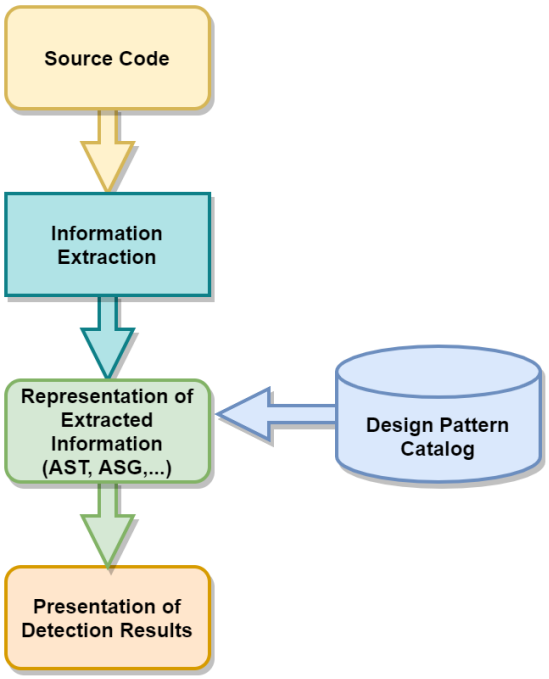


Figure 1: Main steps of structural design pattern detection[3]

While structural reverse engineering solutions can efficiently recognize design patterns. They are however unable to distinguish structurally similar patterns, which often only differ in their behavior[1, 4]. Machine learning techniques, dynamic methods, and static program analysis can be used in behavioral reverse engineering approaches to distinguish these patterns as well. Reverse pattern engineering yields the best results when structural and behavioral approaches are combined.

2.1 GoF pattern reclassification

The GoF book defines 23 design pattern which are classified in three categories based on their purpose; creational, structural, or behavioral. Reference [1] argues that while this categorization of patterns is useful for programmers, it is not helpful for reverse engineering. For that reason a new categorization based on pattern structure and behavior is introduced that is more suitable for reverse engineering purposes. This new categorisation divides the GoF patterns into five categories: language-provided patterns, structure-drive pattern, behavior-driven patterns, domain-specific patterns, and generic concepts. The detection mechanism that is used in Pinot excludes support for language-provided patterns, domain-specific patterns and generic concepts. For that reason these categories will not be further detailed here either.

2.1.1 Structure-driven patterns

The patterns in this category can be identified by mapping inter-class relationships. These relationships support the structuring of the system's arhcitecture without directly influencing the system's behaviour. Inter-class relations could for example separate class responsibility, thereby decoupling the dependencies of classes in a system. The GoF design patterns that are reclassified as structure-driven patterns in the reverse engineering categories are:

- Bridge
- Composite
- Adapter
- Facade
- Proxy
- Template Method

- Visitor

2.1.2 Behavior-driven patterns

Patterns that are designed to influence the behavior of a system fall into this category. This includes creational or structural GoF patterns that use specific behavior to achieve the purpose structure or behavior. This reverse engineering category includes the following GoF patterns:

- Singleton
- Abstract Factory
- Factory Method
- Flyweight,
- Chain of Responsibility,
- Decorator
- Strategy
- State
- Observer
- Mediator

2.2 PINOT

PINOT is a fully automated pattern detection tool that can recognize design patterns in java applications[2]. PINOT supports the reverse engineering of the structural and behavioral-driven categories as defined in section 2.1. PINOT is founded on Jikes[4], an opensource Java compiler written in C++, and extends this with pattern analysis functionality. It does so because compilers construct symbol tables and ASTs, perform semantic checking, and report errors about the source codes. These features are very helpful in determining inter-class relationships and static behavioral analyses.
The detection process that PINOT uses for a given pattern start with identifying the most commonly used features of the given pattern. This pruning approach reduces the search space by skipping over the least likely classes. By focusing on the most commonly used implementations PINOT's efficiency is increased while reducing accuracy of recognizing less commonly used pattern implementations. As an example PINOT's data-flow analysis analyses only singleton patterns that use a boolean type for the flag variable that guards the lazy instantiation. PINOT only uses inter-procedural data-flow analysis for patterns that often involve method delegation.

2.2.1 Build and Execute

To build the Pinot project simply run the following commands.

- cd pinot
- ./configure –prefix=PREFIX –enable-debug
- make
- make install

Note: The **PREFIX** should be the absolute path and not the reference path.

The troubleshooting guide for building the Pinot application can be found in the appendix section A.1.
Syntax to run the Pinot application is as shown below,

```
pinot [options] [@files] file.java
```

[4] http://jikes.sourceforge.net/

For the test scenario, we have picked the following designPatterns git repository that contains various example java implementations of the Gang of Four design patterns.
https://github.com/premaseem/designPatterns
Running Pinot on the repository .java files yielded the following results as shown in Figure 2. We observe that Pinot provides a detailed console output for each .java file analyzed and a statistical report (Pattern Instance Statistics) that provides an overview of the count of GoF patterns observed in the analyzed source code.

We will use this statistical summary as a reference or ground truth to validate the correctness of the Pinot application after refactoring.

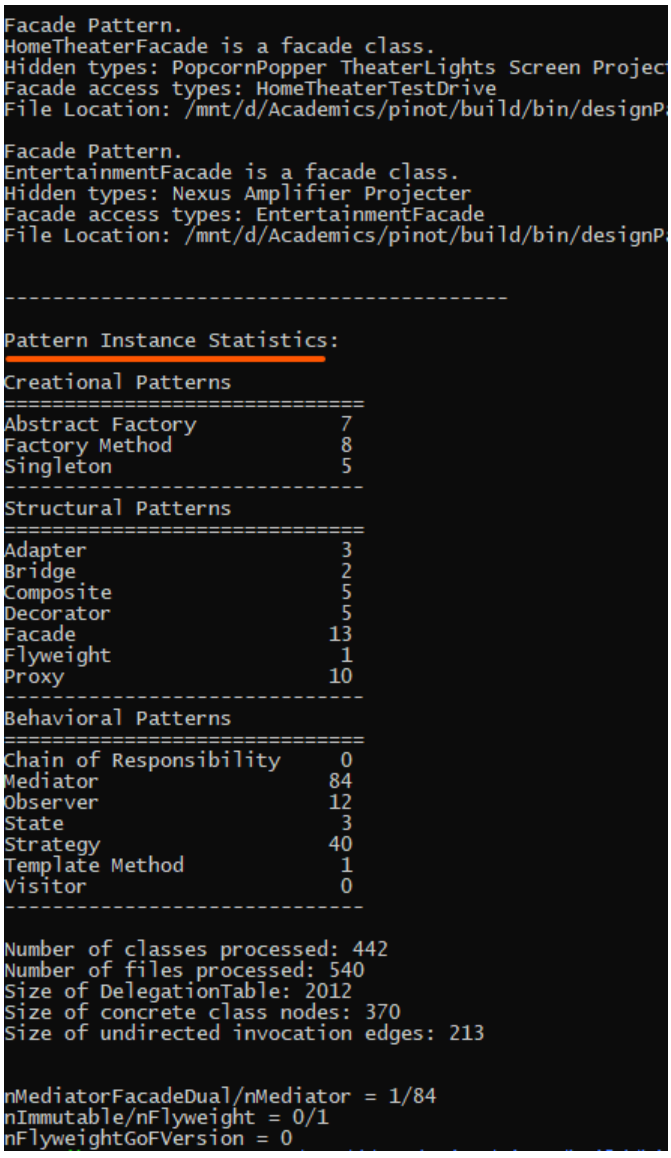


Figure 2: Pinot output before refactoring.

3 Analysis

Todo: Break analysis into 2 sections

This section will first explain how the Pinot source code has been analysed followed by the analysis strategies and outcomes. The reverse engineering application Structure101 has been used to analyse PINOT.
Structure 101 originally provided a generic reverse engineering tool that based its C++ analyses on a second analyses tool called Understand[5]. Structure 101 just released a new beta version of their tool specificly designed for analysing C/C++ applications. We used both tools to analyse PINOT and the high over results from both will be analysed. After these high over overviews, the new Structure 101 version for C/C++ will be used to provide more detailed analyses on PINOT.

3.1 Preparing the Source code for Analysis

The application Structure 101 C/C++ will be used to understand the source code of Pinot. The Clang compiler provides options that create ast-dump files of a C++ project. Compiling PINOT, however, turned out to be more complicated then initially expected. In order to compile PINOT, the original source code has slightly been modified. These modifications are decribed in appendix A.1. The Clang compiler directive ast-dump has been used to generate the AST files[6]. The code listing below shows the complete command used to compile PINOT and generate the corresponding AST files.

```
clang++ -c
        -ferror-limit=0
        -fno-delayed-template-parsing
        -fno-color-diagnostics
        -Xclang
        -ast-dump
        <filepath>.cpp > <ast_output_file_path>.ast
```

Listing 1: Generate AST dump from CPP files.

The AST dumps generated for the different .cpp files are placed in a single folder. The **cpaparser** provided as a build-tool by structure 101 is utilized to translate the AST files into a cpa file that can be used by the Structure 101 application to visualize the source project. The bash command used generate the cpa files are is as follows,

```
java -jar structure101-parser-ast-cpa.jar
          generate-cpa
          -use-db
          -gzip-compress
          -keep-going
          -ignore-compilation-errors
          -d <Path-to-ast-dump> -o <output-cpa-path>
```

Listing 2: Generate CPA from AST dumps.

The building process has been automated using bash script. The script can be found in our Git repository [7], the contents of the script is as follows,

```
#!/bin/bash

SRC_DIR="/mnt/d/Academics/SoftwareEvolution/pinot/src"
AST_DIR="/mnt/d/Academics/SoftwareEvolution/pinot/src/ast_dump"
CPA_FILENAME="Pinotv1.cpa"
file_ext="cpp"
EXCLUDE_LIST=""
```

[5] https://www.scitools.com/
[6] https://releases.llvm.org/8.0.0/tools/clang/docs/HowToSetupToolingForLLVM.html
[7] https://github.com/Swastik-RUG/SoftwareEvolutionPinot/blob/main/build-tools/script.sh

```
8
9  mkdir "$AST_DIR"
10 mkdir "cap_out"
11
12 for entry in "$SRC_DIR"/*.$file_ext
13 do
14         filename=$(echo "$entry" | rev | cut -d '/' -f 1| rev)
15
16
17         if echo $EXCLUDE_LIST | grep -w "$filename" > /dev/null;
18         then
19           echo "Ignored :$filename....."
20         else
21           echo "processing :$filename....."
22           clang++ -c -ferror-limit=0
23                      -fno-delayed-template-parsing
24                      -fno-color-diagnostics
25                      -Xclang
26                      -ast-dump
27                      "$entry" > "$AST_DIR/$filename.ast"
28         fi
29 done
30
31 java -jar structure101-parser-ast-cpa.jar
32                         generate-cpa
33                         -use-db
34                         -gzip-compress
35                         -keep-going
36                         -ignore-compilation-errors
37                         -d "$AST_DIR/" -o "cpa_out/$CPA_FILENAME"
```

Listing 3: Generate CPA from AST dumps.

3.1.1 Instructions on how to use the script

Open the script and change the directory paths and the change the following directories.

- **SR_DIR** -> Pinot source file location.
  Sample: SRC_DIR="/mnt/d/Academics/SoftwareEvolution/pinot/src"

- **AST_DIR** -> A temporary storage to create the AST files.
  Sample: AST_DIR="/mnt/d/Academics/SoftwareEvolution/pinot/src/ast_dump"

The script also provides an exclude list **EXCLUDE_LIST**="Your file names" to exclude any files from the compilation.

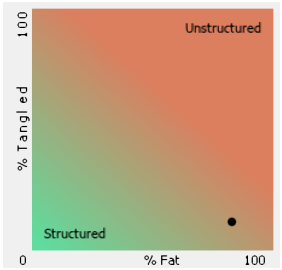
The troubleshooting guide related towards building the **.cpa** files from the **C/C++** source code can be found in the appendix Section A.2.

3.2 Structure 101

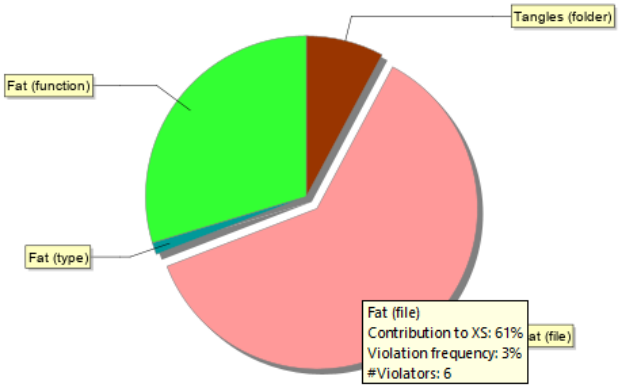
This section describes the architecture of PINOT using Structure 101. Firstly the new beta version Structure 101 C/C++ will be used to analyse PINOT. Thereafter the original way of analysing C++ applications of Structure 101 is used, which utilizes Understand and the generic reverse engineering engine of Structure 101. Finally the analyses results of Understand will be described as well.

3.2.1 Structure 101 C/C++

This section highlights the analyses results of Structure 101 C/C++, which visualize and indicate the architecture and design patterns of PINOT.The application also provides sophisticated visualization aid to track down the components of the source code that can be improved. By utilizing this we determine that the structural complexity of PINOT is **Fat** and **Structured**, as is shown in the Figure **??**.



((a)) PINOT structural over-complexity.



((b)) PINOT structural over-complexity breakdown by metric (and scope).

Figure 3: PINOT files by complexity and size.

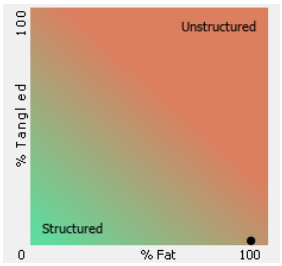
Item	Value
src/ast.h	814
src/ast.cpp	437
src/symbol.h	196
src/control.cpp	164
src/unparse.cpp	162

Figure 4: PINOT fat files.

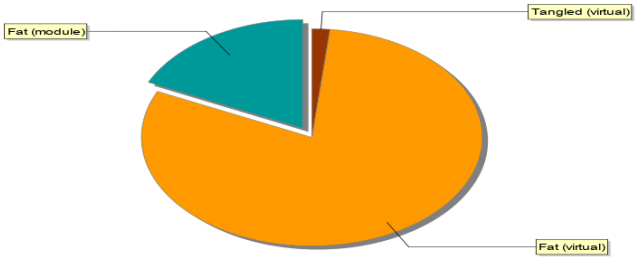
From the figure 3 we can observe the complexity of the PINOT source code broken down into its respective scope and metric. We see that a major portion (61%) of the code is listed as fat files. From figure 4 we can observe that the **ast.h** and **ast.cpp** has the highest contribution towards the Fatness of the PINOT source code. With the penultimate file **symbol.h** only accounts for 25% of **ast.h**. Indicating that, refactoring the **ast.h** and **ast.cpp** will elevate a major portion of the fat file bulk in the PINOT source code.

3.2.2 Structure 101 generic

Now the highlights in the architecture and design pattern of PINOT that will be analysed using Structure 101 generic. Besides describing the analyses results of Structure 101 generic, this section will also indicate the differences between this analyses and the analyses of Structure 101 C/C++. For that reason this section contains similar graphical representations as section 3.2.1.



((a)) PINOT structural over-complexity.



((b)) PINOT structural over-complexity breakdown by metric (and scope).

Figure 5: PINOT files by complexity and size.

Similar to the analyses results from section 3.2.1, we notice that PINOT can be categorized as a **Fat** and **Structured** application. Although the analyses shows a 0 percentage for tangled source code contradictory to 10% indicated by Structure 101 C/C++. The results are graphically represented in figure 5

By comparing figure 6 with figure 4 we see resemblance in the fattest file of PINOT, **ast.h**. The files **ast.cpp** and **control.cpp** also occur in both lists of fat files. For the remaining files it is difficult to say why they appear on a single and not on both lists. Nevertheless we can say with certainty that the files **ast.h**, **ast.cpp**, and **control.cpp** are fat files that favour refactoring.

Item	Value
src/ast.h	1,986
src/class.h	414
src/bytecode.cpp	304
src/ast.cpp	184
src/control.cpp	169

Figure 6: Pinot fat files.

3.2.3 Scitools Understand

In order to analyse PINOT with Strucutre 101 generic, the Scitools Understand analyses tool has to be used to generate an abstraction DB of PINOT's structure. Understand provides nice analyses insights as well. These insights are described in this section.

The PINOT source code comprises of 4416 functions, 107465 lines, 317 classes and 77 files. From the below Figure 7, we can see that **16.78%** of the source code comprises of comments. This is a favourable scenario a good source is considered to contain at least 15% of code comment[8].

[8]https://everything2.com/title/comment-to-code+ratio

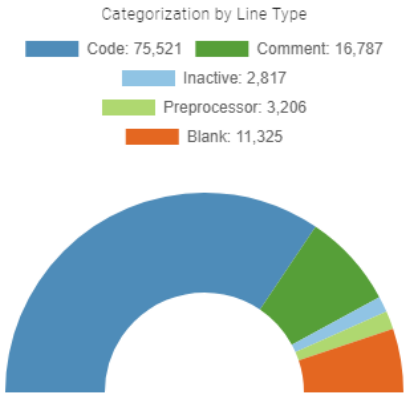
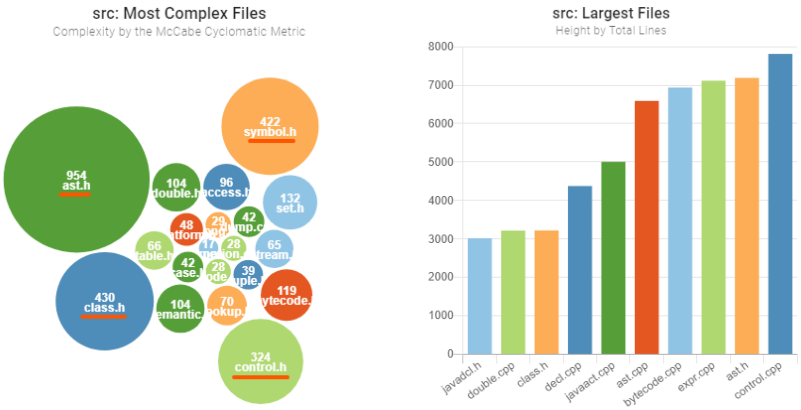


Figure 7: Pinot line breakdown.



((a)) Most Complex files.

((b)) Largest files.

Figure 8: PINOT files by complexity and size.

From figure 8, although the control.h is the largest file in the PINOT source code, it is not the most complex class. The ast.h file is the second largest and the most complex file in the source code. The charts convey that it is worthwhile looking into ast.h, class.h, control.h and symbol.h files as they make up for the majority of the complexity in the source code.

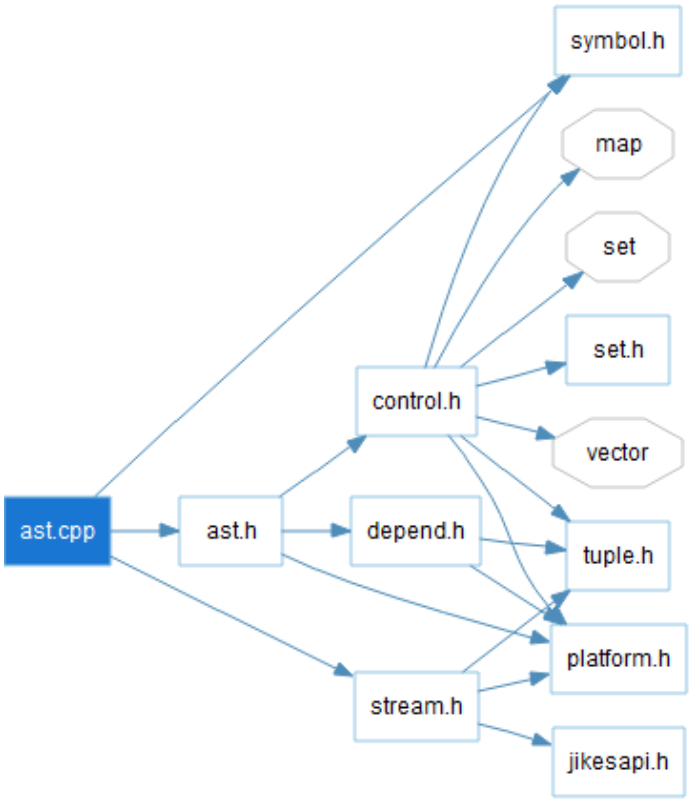


Figure 9: ast.cpp dependency graph.

Figure 9 graphically represents the dependencies of ast.cpp. A quick look at the dependencies of ast.cpp reveals that it is dependent on the 2 other complex classes namely control.h and symbol.h classes. This pattern indicates that precise decisions and care should be taken while refactoring these classes.

3.3 Composition and Cohesive Clusters

The Pinot composition from the root node is as shown in figure 10, the java.g file is the entry point where it initializes the Pinot environment and invokes the respective function calls to start the GoF pattern analysis.

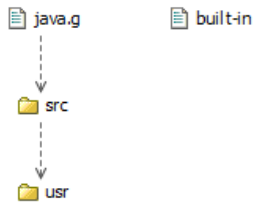


Figure 10: Pinot source code root composition.

From the figur 21, we can clearly observe 2 major clusters marked in **blue**. The cluster towards the left houses 3 sub-clusters marked in **red**, **turquoise** and **green**. The source code effectively contains 4 cohesive clusters marked in red, turquoise, green and the blue cluster towards the right.

From the clusters and the naming choice used to define the files. We can roughly guesstimate[9] that the cluster on the right marked blue relates to the programs entry point, and configurations. Whereas the clusters red, turquoise, and green comprises of the entire Pinot framework.
The **usr** folder contains the includes and libraries required by Pinot. The C++ compilers, gcc, llvm-10, and other supplements based on the Pinots system requirements is clustered in this folder as shown in figure 11.

Figure 11: Structure 101 Pinot source code usr folder cluster.

3.4 Architecture Diagram

The architecture for Pinot was generated using the Structure 101 views. The resulting structured diagram can be viewed in figure 22. The code groups are color coded as per their groups or clusters. On performing an Top-down analysis and reading the code we attempted to determine the general intent of each of these clusters.

- The classes marked under the green boxes contain utility, constants and configurations. The gof.cpp class is especially interesting since it contains many utility classes that are used by Pinot to deconstruct and understand the java code.
- The one's marked in blue serve as the backbone of the project. The implementations generally relate to the data flows, data types, scanners and table. These files include the basic structure of Jikes that PINOT was build upon. For example the files jikes.h and jikes.cpp are easily recognized. The project has implemented custom double and long data types to improve the accuracy of these values when a cast from string data type is performed. As a note, the double precision is also not reliable in C/C++ as they are normally represented using a finite precision binary format.
- The classes marked in yellow and orange make up for most of the framework code which Pinot is constructed with.

3.5 Dependencies

The dependency graph that is shown in figure 23 divides the application in 3 main clusters and 5 assisting files. The dependency structure shows clusters and assisting files similar to the architecture diagram that is discussed in subsection 3.4. Contrary to the architectural diagram the dependency diagram indicates the presence of 3 tangled clusters. The influence of these clusters is little as they consists either of 2 or 3 files.

[9] An estimate based on a mixture of guesswork and calculation.

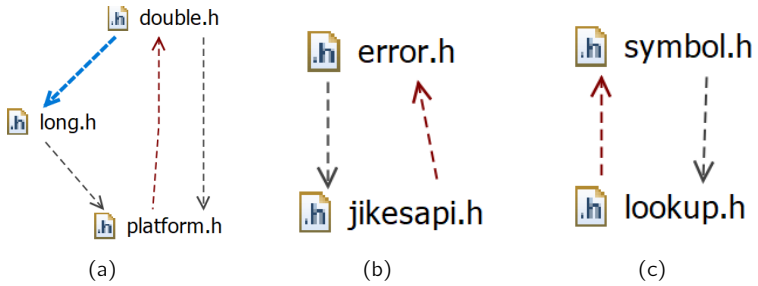


Figure 12: Tangled clusters in PINOT.

The largest tangled cluster consists of the files *double.h*, *long.h*, and *platform.h*. As figure 12 indicates the classes in *double.h* extends classes from *long.h*. And both *double.h* and *long.h* use type definitions from *platform.h*. Both the 2 smaller tangled clusters are only marked as tangles because of friend classes. classes that are defined in *jikesapi.h* and *lookup.h* befriended classes in respecitevely *error.h* and *symbol.h*.

3.6 OOMetris Analysis

The Class OO Metrics Report provides the following object-oriented metrics for each class that has been analyzed [5]:

- LCOM (Percent Lack of Cohesion): 100% minus the average cohesion for class data members. A method is cohesive when it performs a single task.
- DIT (Max Inheritance Tree): Maximum depth of the class in the inheritance tree.
- IFANIN (Count of Base Classes): Number of immediate base classes.
- CBO (Count of Coupled Classes): Number of other classes coupled to this class.
- NOC (Count of Derived Classes): Number of immediate subclasses this class has.
- RFC (Count of All Methods): Number of methods this class has, including inherited methods.
- NIM (Count of Instance Methods): Number of instance methods this class has.
- NIV (Count of Instance Variables): Number of instance variables this class has.
- WMC (Count of Methods): Number of local methods this class has.

We will first begin analysing the OO Metrics of Pinot as a whole, followed by the individual analysis of the fat files that were found in the Structure 101 C/C++ as shown in table 4. The analysis should provide us with insights as to why these files are fat in a Object-oriented design perspective. We will plot a pi chart to understand which OO Metrics has the highest influence and pick the significant ones for a deeper analysis.

3.7 Pinot Objected-Oriented Metrics Analysis

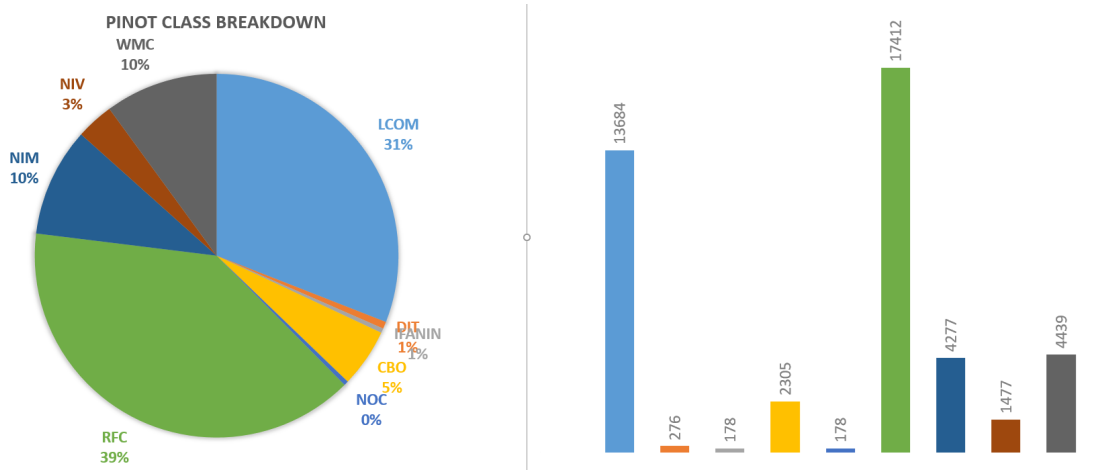


Figure 13: PINOT OO Metrics breakdown.

The figure 13 shows the object-oriented Metrics breakdown on the entire Pinot project. From the pi chart we see that 31% accounts for lacks of Cohesion (LCOM), displays high count of methods at 39% or 17412 (RFC), and 10% count of Methods (WMC) and Instance methods (NIM) that is a already a part of RFC.

Further analysing the LCOM and RFC should provide us with some insightful results. For this we determined the top 10 classes that are responsible as the highest contributors towards these Metrics as shown in figure 25 and 24. One of our primary goal is to determine the cause for Pinot being fat, and this can be explained by the high percentage of RFC metrics in comparison to the rest. The classes in Pinot have too many methods in them, splitting these methods logically should reduce the fatness of the project. The top 10 WCM and NIM can also be found in figure 27 and figure 26. The WCM and NIM would point to the same method or class, since a method declared in C/C++ is always instanced by the compiler, and are thus same.

3.7.1 Analysing the Ast Class

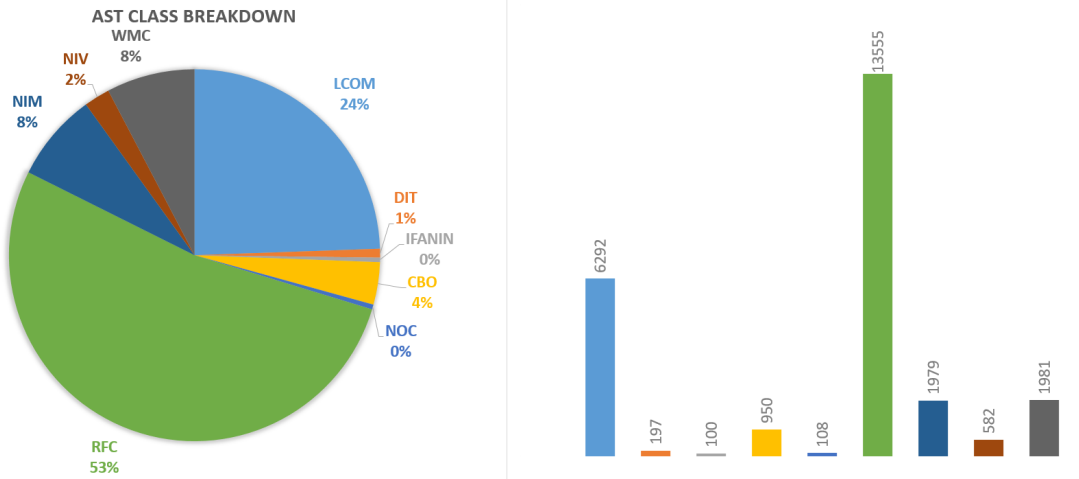


Figure 14: PINOT Ast class OO Metrics breakdown.

From figure 14, similar to our Pinot project analysis, the AST class analysis also reveals that the package lacks cohesion and have a high count of methods, LCOM (24%) and RFC (53%) respectively. The RFC is the Ast package is especially significantly large compared to the rest of the classes and Pinot as a whole. So it is safe to assume that most of the RFC bulk resides in the Ast package.

The top 10 LOCM and RFC classes for the Ast package can be found in figure 29 and figure 28. WCM and NIM were left out from deeper analysis as they only contribute towards 8% of the bulk.

3.8 Analysing the Symbol Class

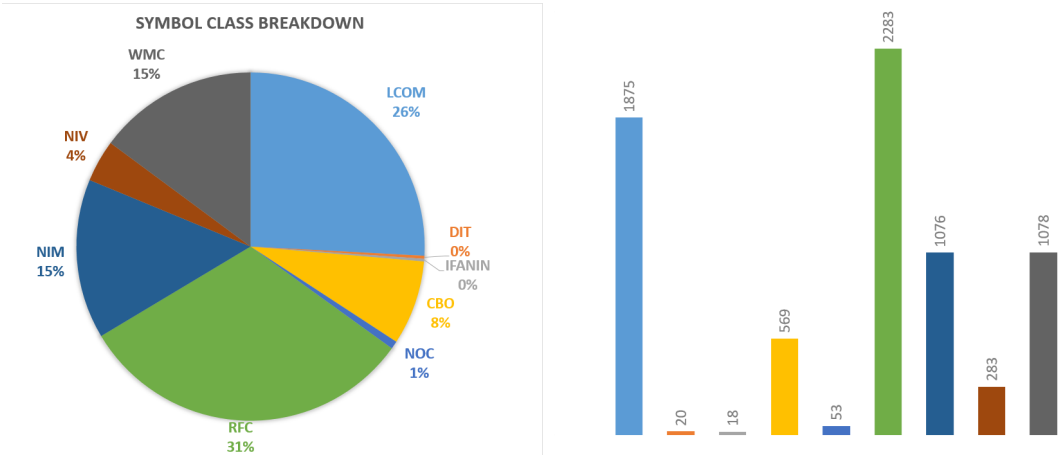


Figure 15: PINOT Symbol package OO Metrics breakdown.

As expected from figure 15, the bulk resides in the LOCM (26%) and RFC (31%), but in comparison to the Ast package the values recorded are relatively less. Also the WMC and NIM contribute towards 15% of the metrics count which would mark them for a deeper analysis.
The top 10 LCOM, RFC, WMC, and NIM can be found in figures 31, 30, 33 and 32. If you look at these images closer we can see that most of the classes responsible for the bulk in RFC, LCOM, WMC and NIM are due to the **Ast** classes itself. A detailed check has to be performed separated to determine if the Ast classes are duplicated to **Symbol** class or is this due to method invocations to the Ast package. The latter would be ideal as reducing the fatness of Ast package would passively reduce the fatness of the Symbol class.

3.9 Analysing the Control Class

The significant portion of the OO Metrics is again comprised of the LOCM (27%) and RFC (34%), some minor contribution from the WMC (13%) and NIM (13%) as shown in figure 16.

The top 10 LCOM, RFC, WMC, and NIM can be found in figures 35, 34, 37 and 36. On observation we see that the class **Semantics** has major contribution towards the fatness of the Control class as it amounts for a significant amount towards the RFC, WMC and NIM metrics. The LOCM is influenced by the AstDeclared class but it is closely followed by the StoragePool and Semantics class. Conceptually a Control package should not house the StoragePool class which should ideally have been split, analysis is required to see if StoragePool is duplicated in the Control package.

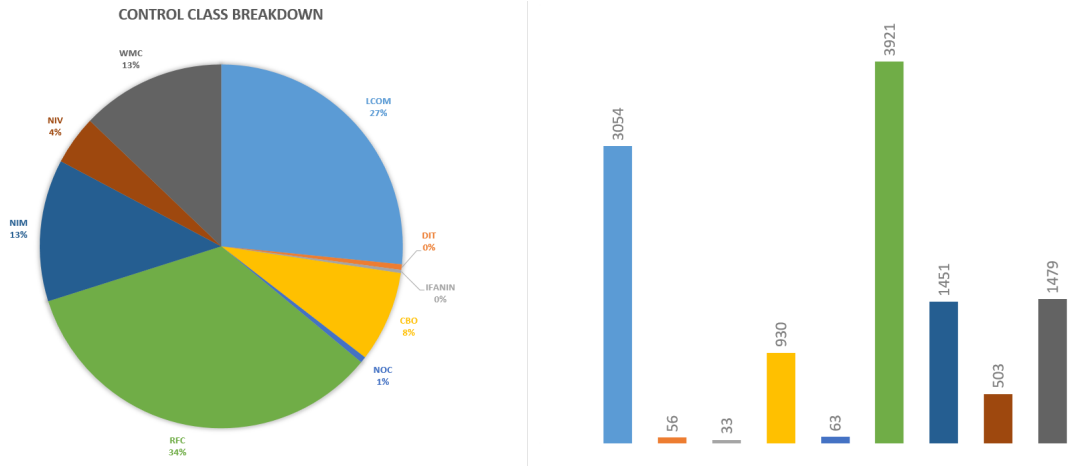


Figure 16: PINOT Control package OO Metrics breakdown.

4 Refactor plan

The conducted analysis of PINOT that is outlined in section 3, shows that PINOT's main quality issue is fat files. Among the source files of PINOT are multiple files that exceed 5000 lines; *ast.h*, *ast.cpp*, *control.cpp*, *expr.cpp*. Apart from fat code files, all files are part of the same namespace and are structured in a single folder. Thereby the file and package structure of PINOT does not provide any clarification on the application structure either. This section will introduce a refactor plan for PINOT that is focussed on improving the file structure and decreasing the file size and thus fatness of PINOT.

4.1 File Structure

Currently the file structure of PINOT is completely unstructured as all the source files are grouped in one single folder. The file structure is the easiest way to indicate a little structure in software project. While PINOT does not contain any file structuring yet, from the Structure 101 analysis we know that PINOT consists of 3 main clusters. Thus it should be possible to split the source files over at least 3 sub folder to create a logical separation of source files based on functionality. The base file structure can be deduced from the dependency graph that is displayed in figure 23. The restructuring of the folder structure will be enhanced more carefully according to the other refactor steps that are explained in the rest of this section.

4.2 Abstract Syntax Tree

From the analysis section, 3, we learned that *ast.h* is the fattest file in PINOT. For that reason we want to refactor *ast.h* and its implementation file *ast.cpp*. It will be refactored according to the following steps:

1. **Unrelated Class** - Move the class `StoragePool` and its methods to a separate file. The `StoragePool` class is used as an storage object that holds all AST nodes. Because it only holds the AST tree nodes, this class is very suitable to be extracted from *ast.h* into its own file *storagepool.h*.

2. **Related Classes** - Extract the different AST classes into their own header file. The file *ast.h* contains the class `Ast`, which is the base node from which all other nodes inherit. All leaf nodes can however inherit again from a more detailed base node. E.g. `AstDeclared` is the

base node for `AstFieldDeclaration`, `AstMethodDeclaration`, etc. These detailed base nodes will be extracted from *ast.h* into their own files.

3. **Extracted Class Methods** - The file *ast.cpp* contains method implementations of some of the classes that are going to be extracted. These methods will also be extracted and moved to a new appropriate implementation file.

4. **Import Consistency** - The newly created (header) files have to be imported in the dependent files.

4.3 Control

From the analysis section, 3, we learned that *control.cpp* is among the fattest implementation files in PINOT. Extensive source code analysis also indicates that control.cpp is highly dependent on *ast.h*. Moreover, the pattern recognition logic is located in the `Control` class, which is defined in the control header and implementation files. In order to decrease the fatness and increase the logical responsibility segregation the control files will be refactored. The refactoring will follow the steps that are described below:

1. **Control Constructor** - The constructor of `Control` is responsible for processing all java files, generating the AST for those files, and analysing the patterns in these files. The functionality is going to be moved to helper methods, to decrease the logic inside the constructor. The responsibilities of the constructor will therefore remain unchanged.

2. **Global Count Variables** - The control implementation file contains global variables to count the occurrences of patterns in the to be analysed java application. These variables are all used by the `Control` constructor to print the number of patterns found after analysis. Apart from that each global variable is incremented by the method responsible for detecting that specific method. The global variable can be moved to the `Control` class as private members and each detecting method should return an integer that represents the number of detected patterns.

3. **Remove Unused Methods** - The control implementation file contains methods that are unused in the analysis. These methods will be removed.

4. **Table Classes** - The control files define 5 different table classes to support the analyses; `ReadAccessTable`, `WriteAccessTable`, `DelegationTable`, `ClassSymbolTable`, and `MethodSymbolTable`. These classes will all be moved to their own header and implementation files to decrease the fatness of the control files.

4.4 Symbol.h

References

[1] Nija Shi and Ron Olsson. "Reverse Engineering of Design Patterns for High Performance Computing". In: *Workshop on Patterns in High Performance Computing (PatHPC'05)* (May 2005). URL: https://www.cs.ucdavis.edu/~shini/research/pinot/reverseJavaPatterns.pdf.

[2] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1995.

[3] Francesca Arcelli et al. "A comparison of reverse engineering tools based on design pattern decomposition". In: *2005 Australian Software Engineering Conference*. IEEE. 2005, pp. 262–269. URL: https://www.researchgate.net/profile/Francesca_Arcelli_Fontana/publication/4129225_A_comparison_of_reverse_engineering_tools_based_on_design_pattern_decomposition/links/542d7ce90cf29bbc126d34f5.pdf.

[4] Lothar Wendehals. "Specifying patterns for dynamic pattern instance recognition with UML 2.0 sequence diagrams". In: *Proc. of the 6th Workshop Software Reengineering (WSR), Bad Honnef, Germany, Softwaretechnik-Trends*. Vol. 24. 2004, p. 2. URL: http://lothar-wendehals.de/publikationen/dokumente/WSR_Wen.pdf.

[5] Understand by Scitools. *Understand Metrics list documentation*. https://support.scitools.com/t/what-metrics-does-undertand-have/66. [Online; accessed 08-December-2020]. 2020.

A Troubleshooting

The section will contain detailed information about the issues faced while working with the project and potential troubleshooting guides to resolve the issue.

A.1 Building Pinot

The troubleshooting guide is related to the issues found while building Pinot using the instructions provided by the original developers. Links to the source code and the installation guide:
https://www.cs.ucdavis.edu/ shini/research/pinot/
https://www.cs.ucdavis.edu/ shini/research/pinot/PINOT_INSTALL

A.1.1 Cannot convert 'bool' to 'State*' in initialization

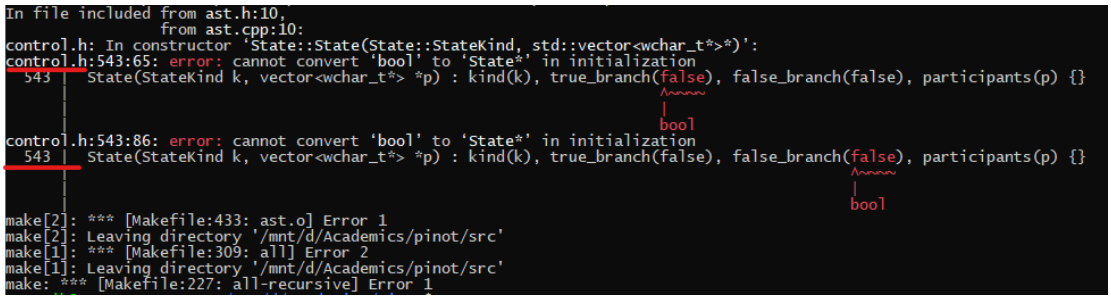


Figure 17: cannot convert 'bool' to 'State*' in initialization Error.

From figure 17, we can observe that the exception or error is raised in control.h file line number 543. We can see that the two parameters true_branch(false) and false_branch(false) are causing the issue.

```
State(StateKind k, vector<wchar_t*> *p) : kind(k), true_branch(false),
                                                   false_branch(false),
                                                   participants(p) {}
```

As the values are false, it wont significant difference if we change it to "NULL" and Changing the values to "NULL" fixes this build issue. You should see the following console log as shown in Figure 18 after applying this fix.

```
State(StateKind k, vector<wchar_t*> *p) : kind(k), true_branch(NULL),
                                                   false_branch(NULL),
                                                   participants(p) {}
```

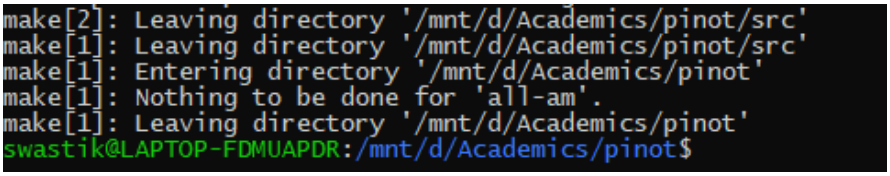


Figure 18: Build success after the fix.

A.2 Building .cpa files for Structure 101

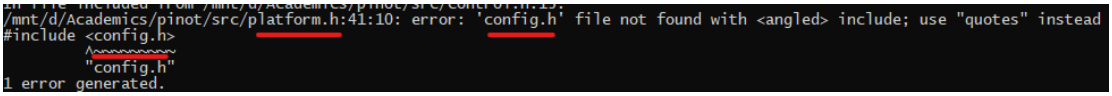


Figure 19: Config.h build error, generating .cpa file for Structure 101.

This is an isolated issue related to platform.h file, where the config.h a custom header file that is declared in angular braces. In the latest make version as of 30-11-2020, this throws an error.

To resolve this issue simply change the **<config.h>** to **<config>** or **"config.h"** in **platform.h** file.

The same issue can be observed for <new.h> in line number 169 and 174 in platform.h file. We will simply change it to <new> (or as declared in line 172).

The other errors and warnings can simply be ignored, unless the script is terminated without completion.

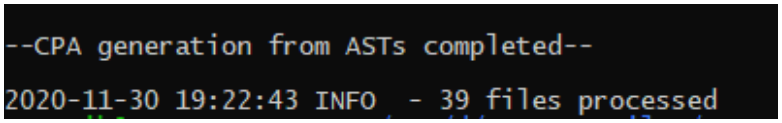


Figure 20: cpa file generated message, after fix to platform.h.

The message displayed in Figure 20 should now be printed on your console once the script has finished execution.

B Version History

Version	Author	Date	Description
1.0	SO	16/11/2020	Add Section 2
		17/11/2020	Add Section 1
	SN	18/11/2020	Added Section 3
2.0	SO	27/11/2020	Added 2.2
		01/12/2020	Revised 3
		01/12/2020	Add 3.2.2
	SN	30/11/2020	Added revision History.
			Added Section 2.2.1
			Figure 1 enhancement.
			Added appendix section A A.1
			Added Section A.2
		1/12/2020	Added Section 3.2.3
			Started Section 3.3

Continued on next page

Continued from previous page

Version	Author	Date	Description
			Started Section 3.4
3.0	SO	3/12/2020	Updated review suggestions.
			Moved big diagrams to appendix C.
			Added section 3.5.
		8/12/2020	Added section 4.
		9/12/2020	Added section 4.3.
	SN	8/12/2020	Working on Section 3.6 and Appendix C.1.
			Image enhancement for Figure 4.
		9/12/2020	Finished Section 3.6 and Appendix C.1.
			Section rework 3.3.

C Figures

C.1 Object-Oriented Metrics Analysis

C.1.1 Pinot OOM Analysis

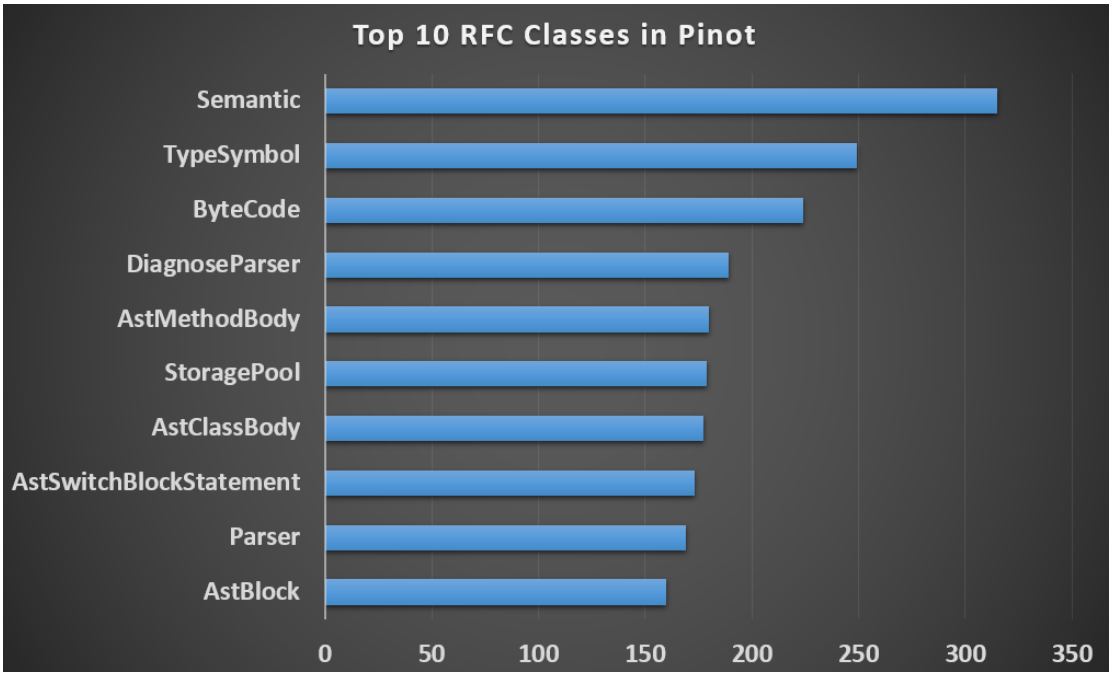


Figure 24: Top 10 RFC classes in Pinot.

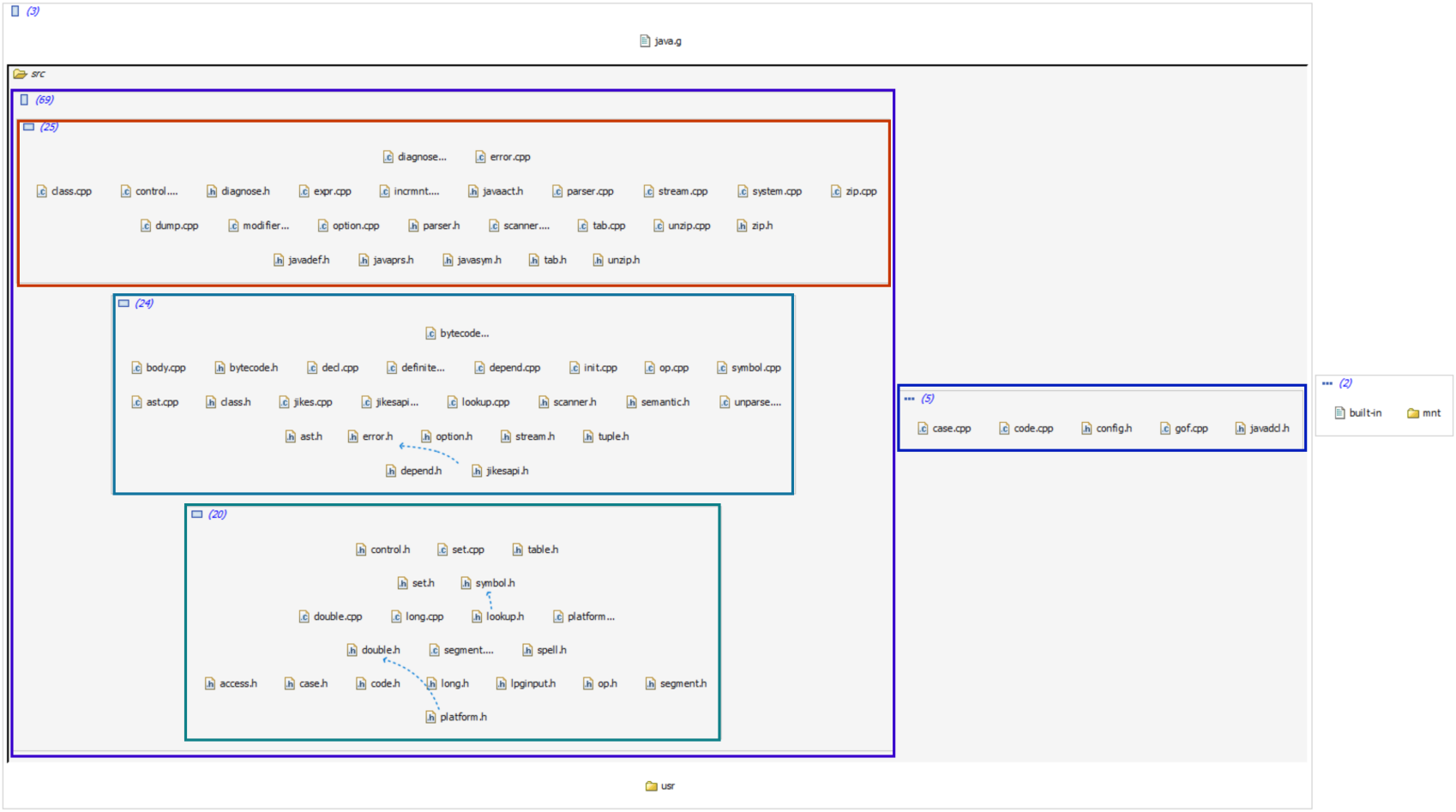


Figure 21: PINOT source code cohesive clusters as reversed engineered by Structure 101.

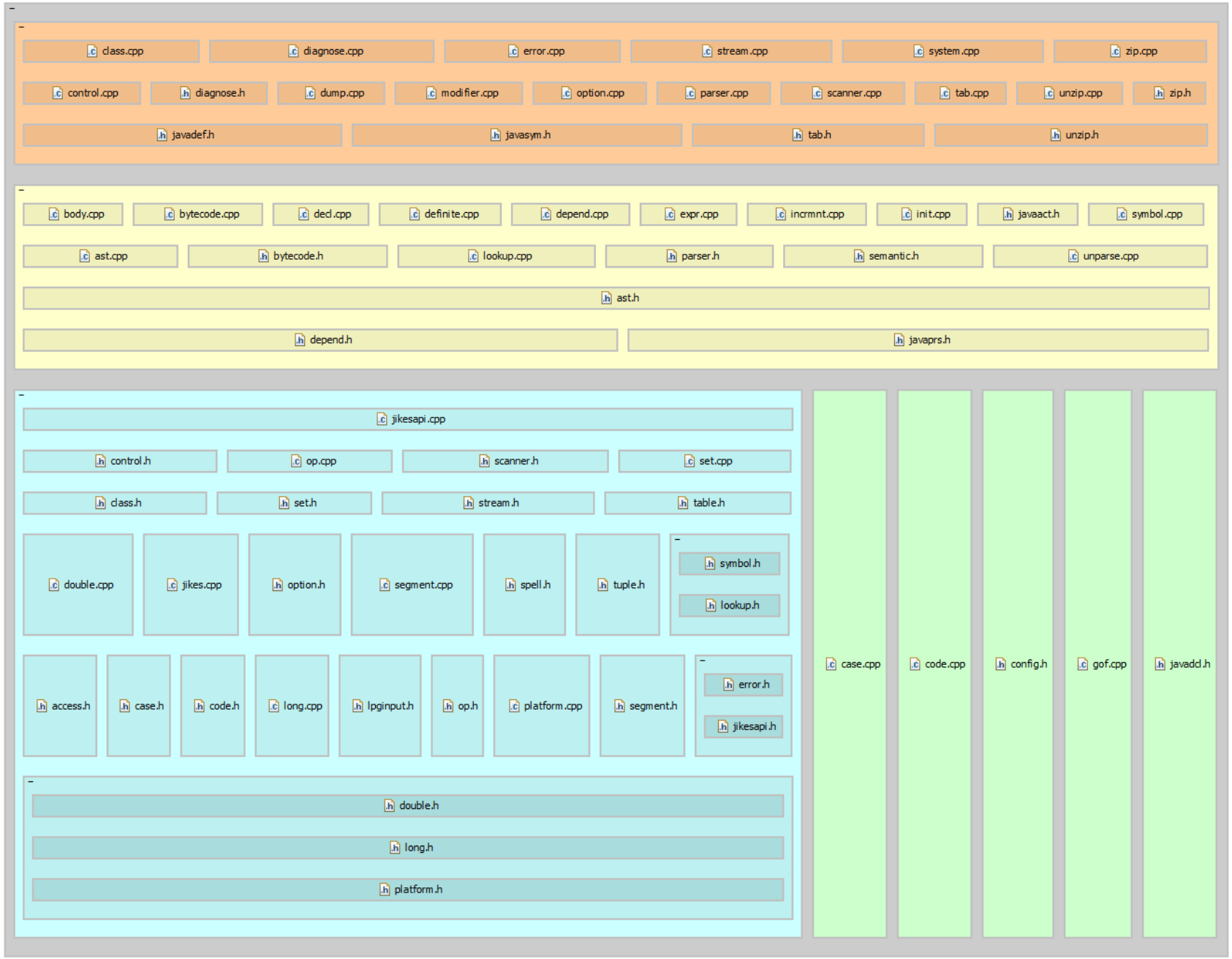


Figure 22: Structure 101 Pinot source code Architecture.

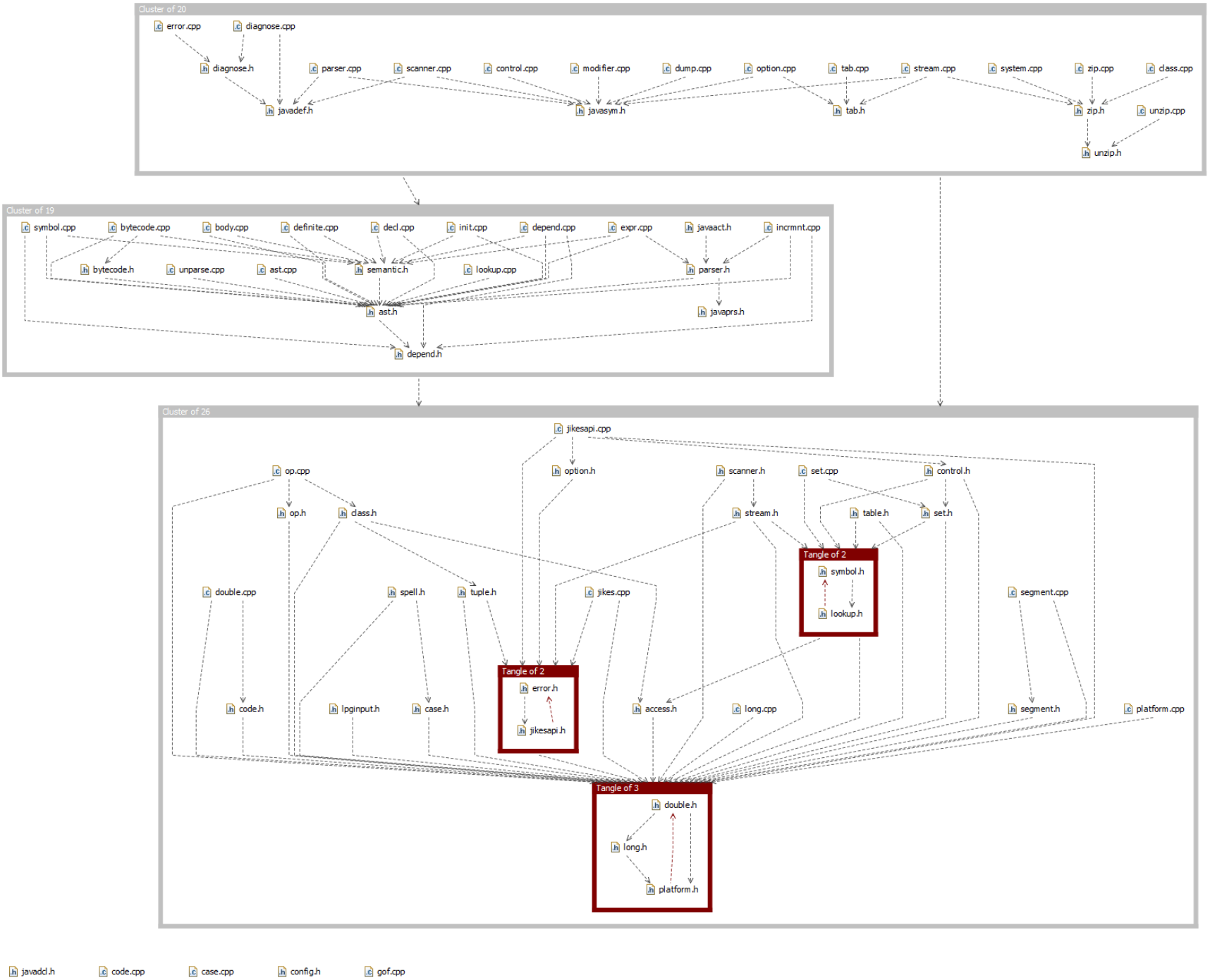


Figure 23: Dependencies between files in PINOT

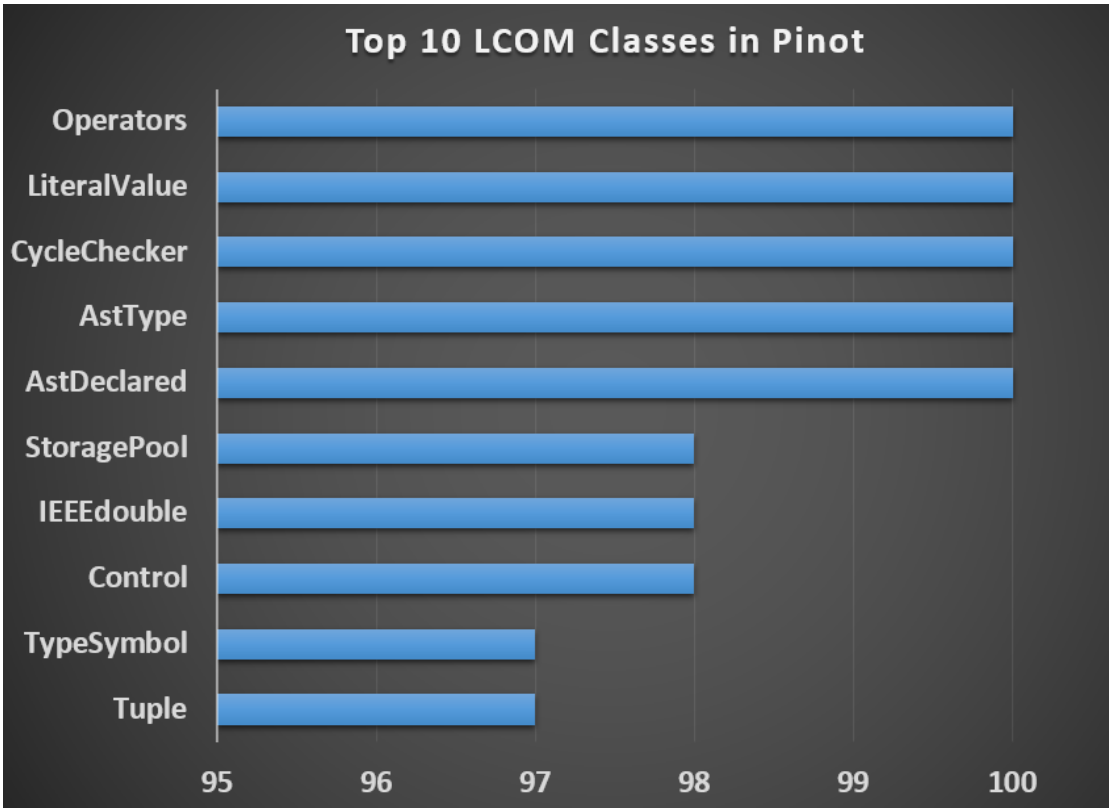


Figure 25: Top 10 LCOM classes in Pinot.

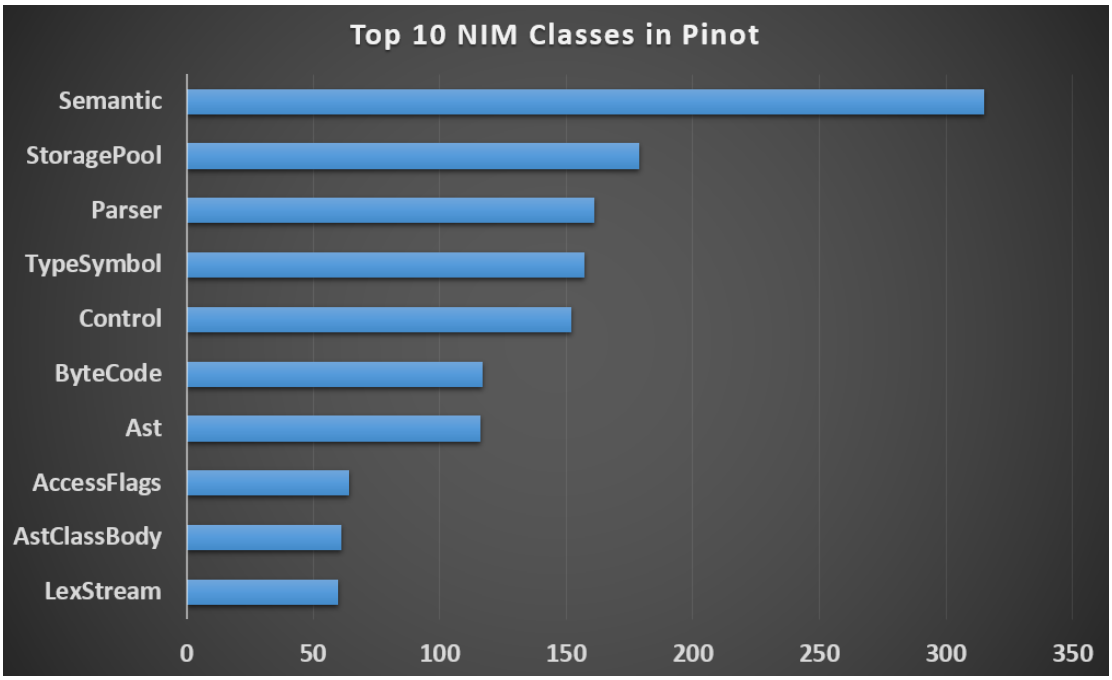


Figure 26: Top 10 NIM classes in Pinot.

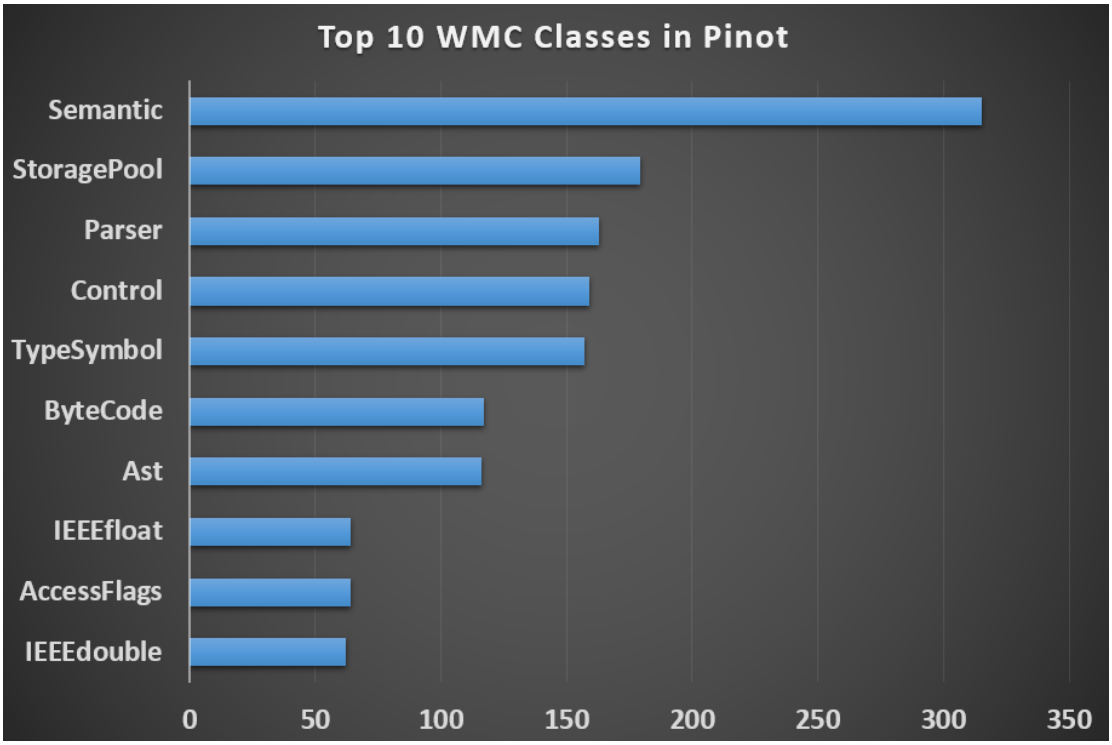


Figure 27: Top 10 WMC classes in Pinot.

C.1.2 Ast OOM Analysis

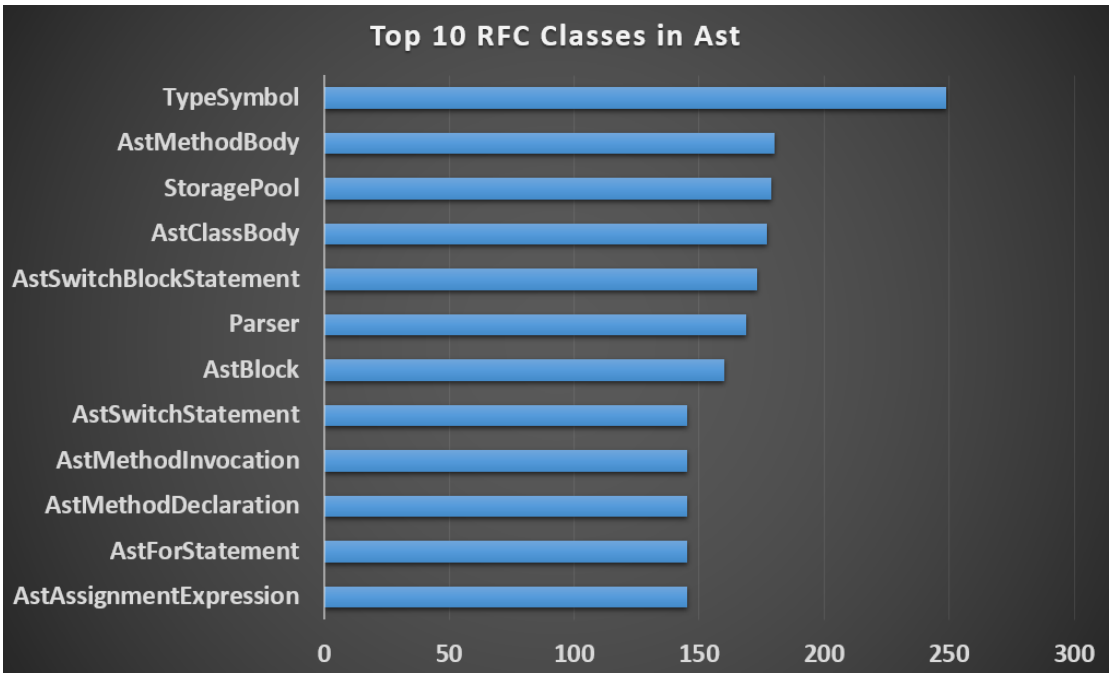


Figure 28: Top 10 RFC classes in Ast package.

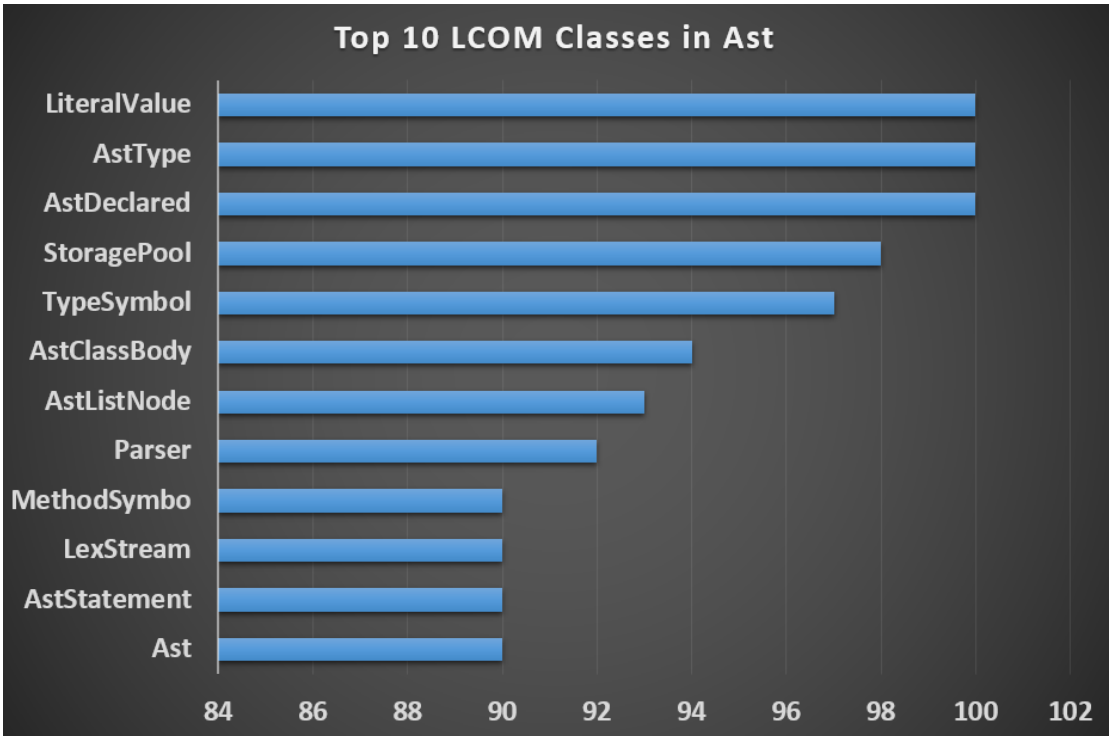


Figure 29: Top 10 LCOM classes in Ast package.

C.1.3 Symbol OOM Analysis

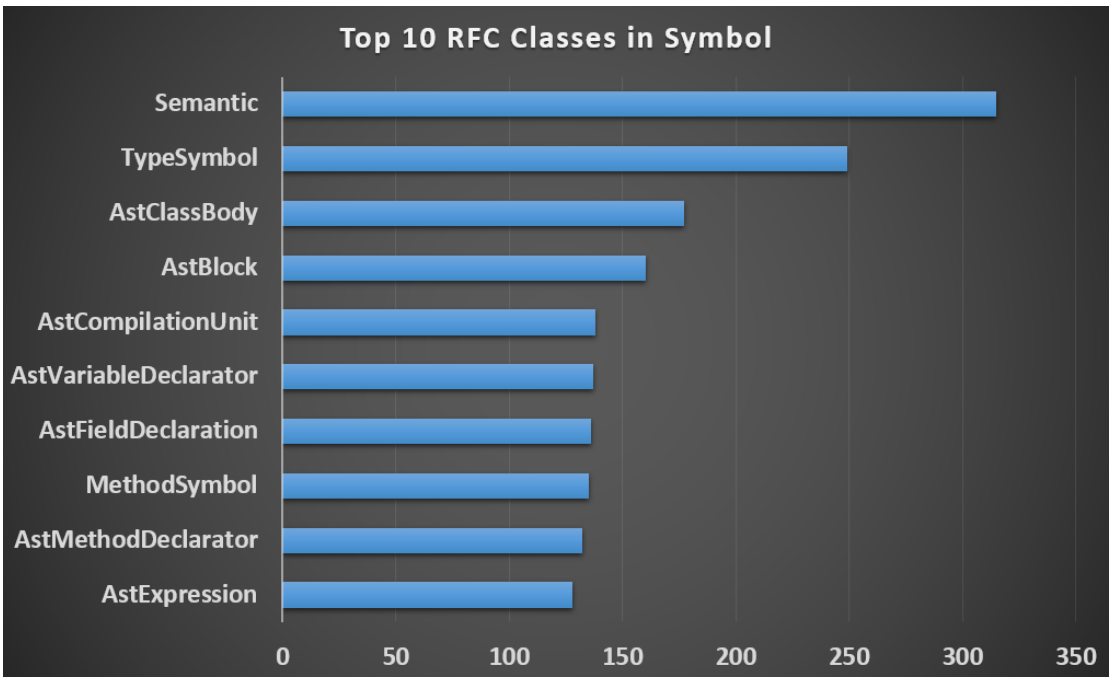


Figure 30: Top 10 RFC classes in Symbol package.

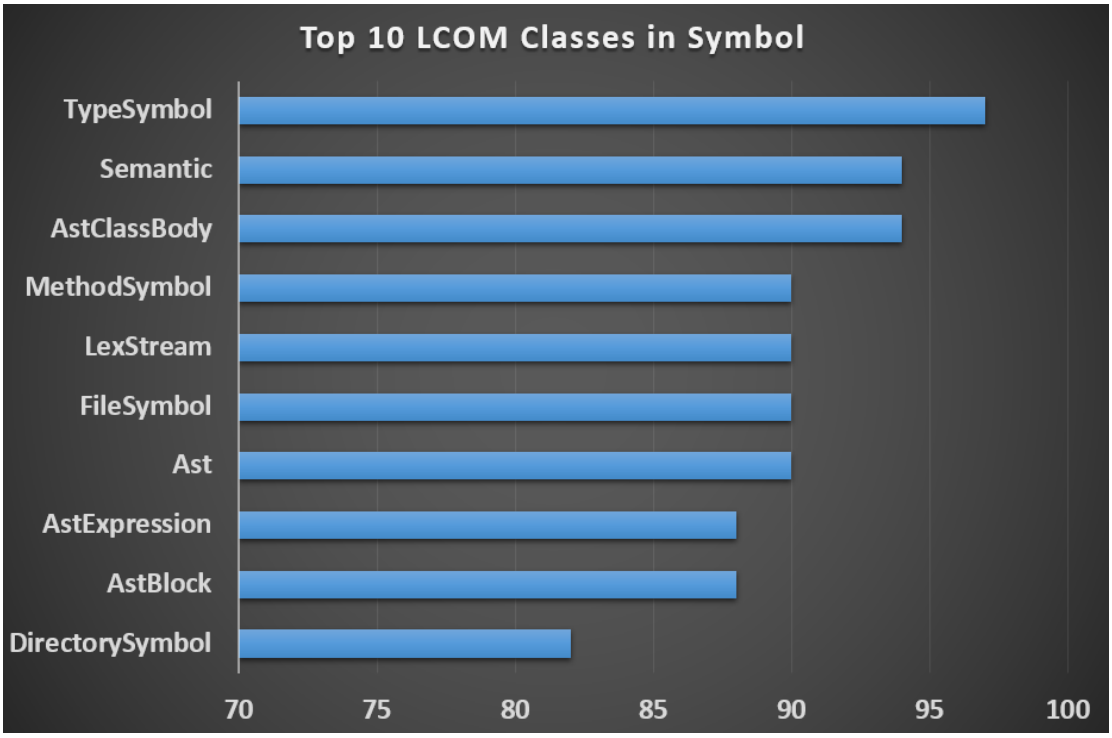


Figure 31: Top 10 LCOM classes in Symbol package.

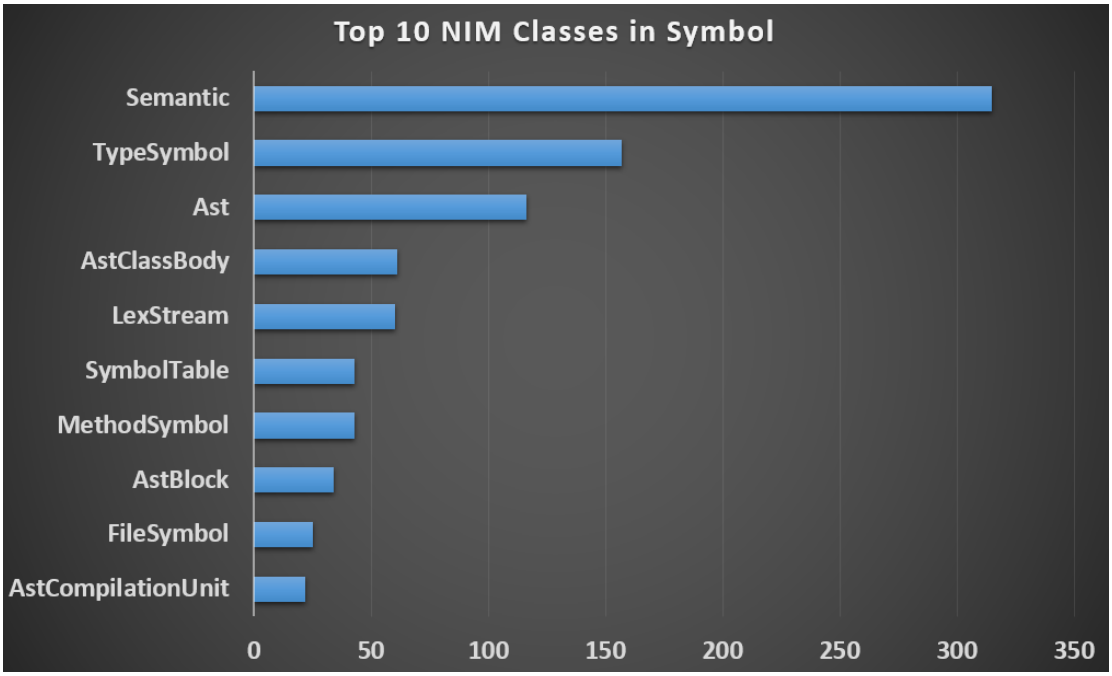


Figure 32: Top 10 NIM classes in Symbol package.

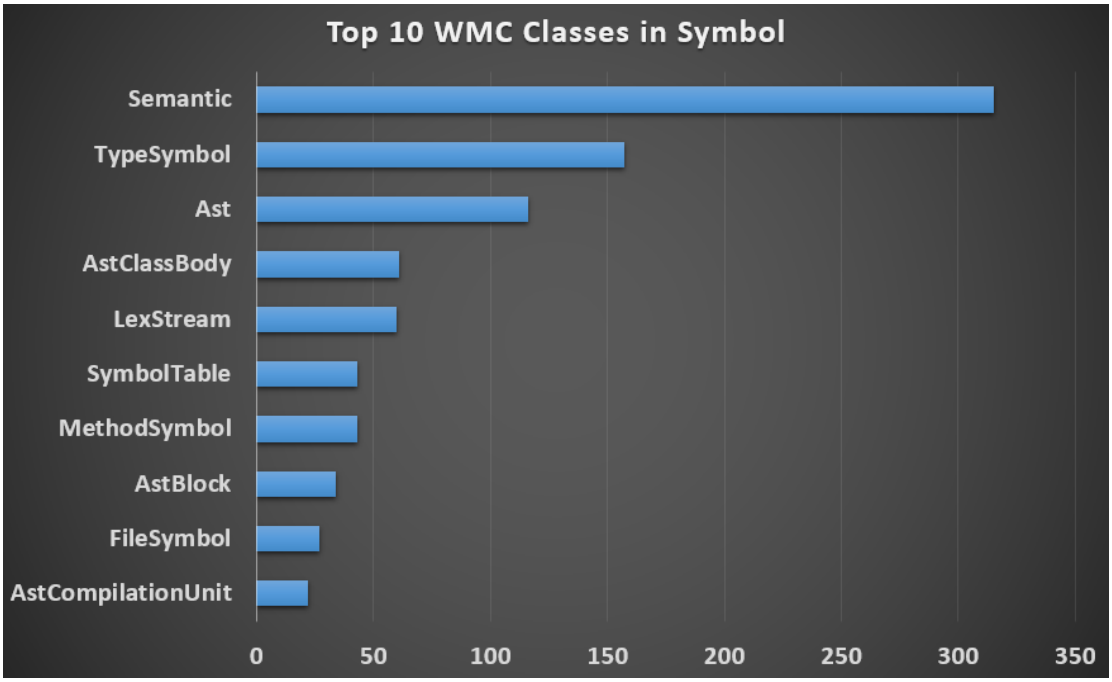


Figure 33: Top 10 WMC classes in Symbol package.

C.1.4 Control OOM Analysis

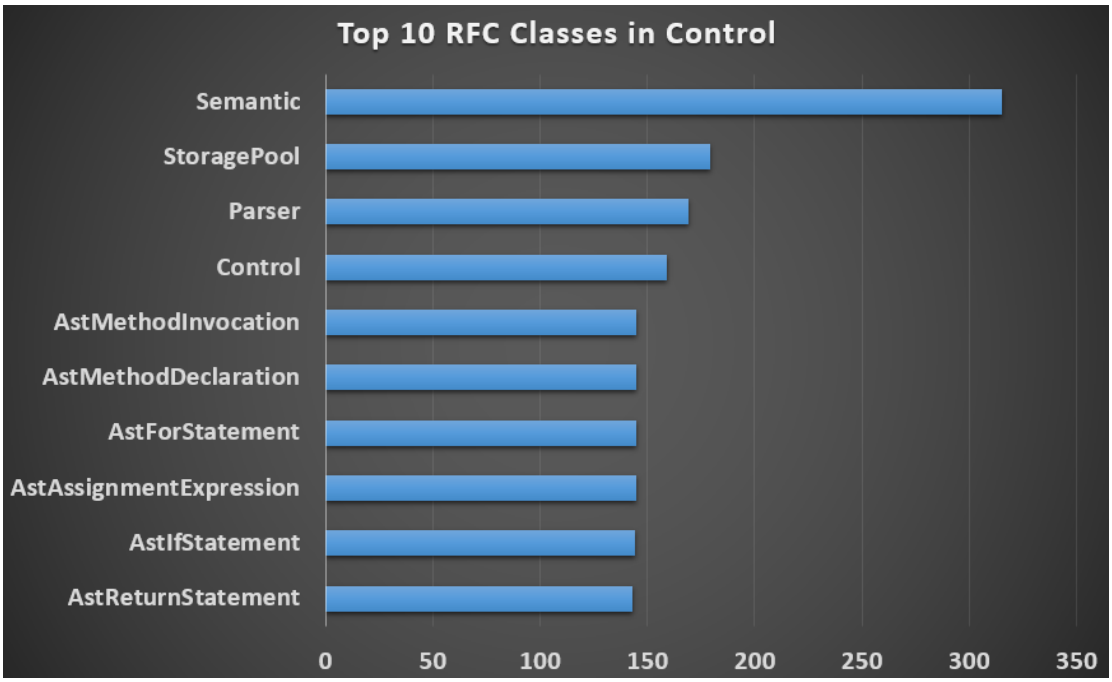


Figure 34: Top 10 RFC classes in Control package.

Figure 35: Top 10 LCOM classes in Control package.

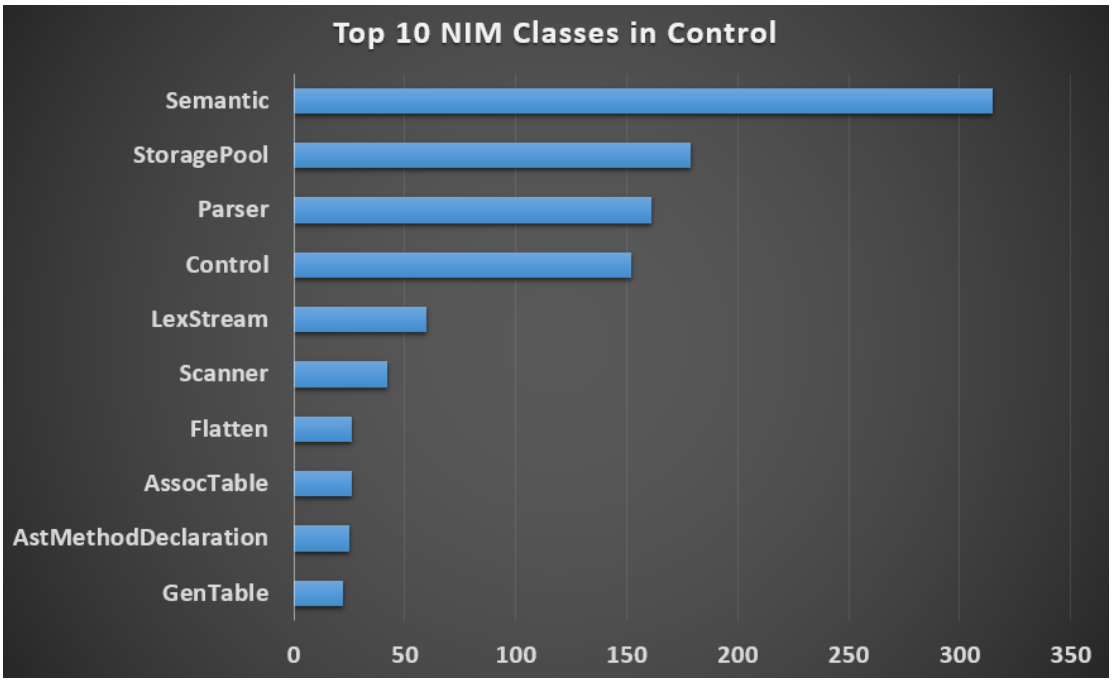


Figure 36: Top 10 NIM classes in Control package.

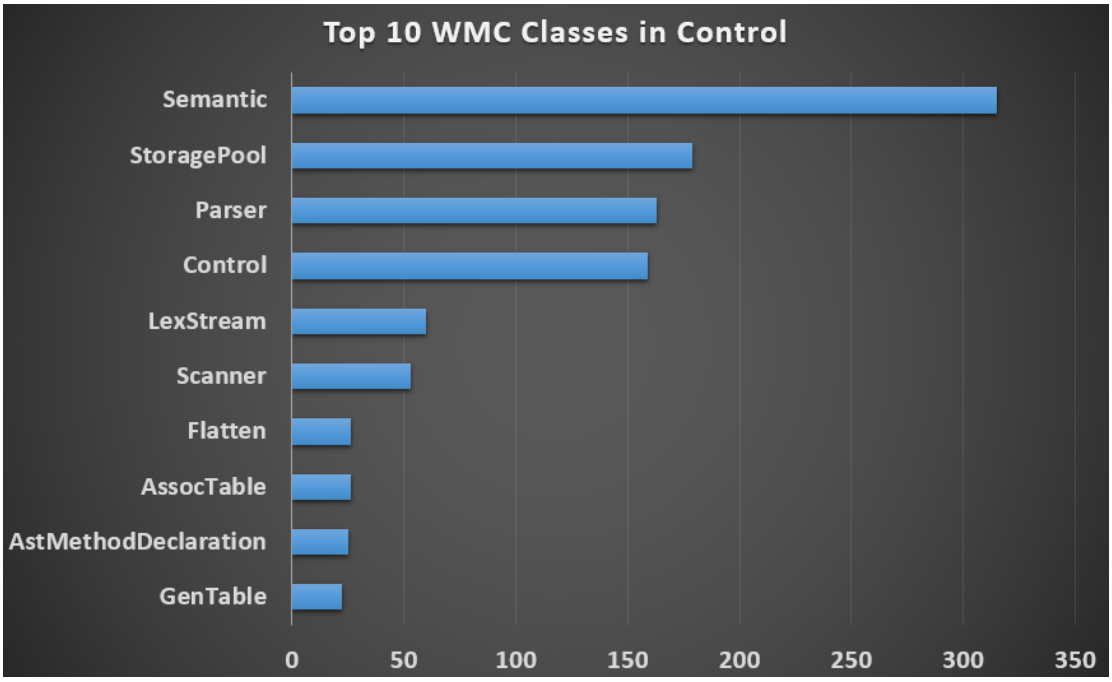


Figure 37: Top 10 WMC classes in Control package.