

**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>

January 17, 2021

Contents

Contents	1
1 Introduction	3
2 Background	4
2.1 GoF pattern reclassification	4
2.1.1 Structure-driven patterns	5
2.1.2 Behavior-driven patterns	5
2.2 PINOT	5
2.2.1 Build and Execute	5
3 Analysis of PINOT	7
3.1 Preparing the Source code for Analysis	7
3.1.1 Instructions on how to use the script	8
3.2 Analysing PINOT using Structure101 tool	8
3.2.1 Structure101 C/C++	8
3.2.2 Structure101 generic	9
3.2.3 Analysing PINOT using Sci-tools Understand	10
3.3 Composition and Cohesive Clusters	12
3.4 Architecture Diagram of PINOT	13
3.5 Dependencies	15
3.6 OO-Metrics Analysis of PINOT	15
3.6.1 PINOT Objected-Oriented Metrics Analysis	16
3.6.2 Analysing the Ast Class	17
3.6.3 Analysing the Symbol Class	17
3.7 Analysing the Control Class	18
4 Component Analysis of PINOT	19
4.1 Understanding the PINOT source	19
4.1.1 Declaration module	19
4.1.2 Generation of Abstract Syntax Tree	19
4.1.3 Orchestration and AST analysis	20
4.1.4 Standalone classes or Unclustered classes	20
4.1.5 Platform cluster	20
4.1.6 Jikes Compiler	21
4.1.7 Symbols cluster	22
4.1.8 Ast code file in PINOT	22
4.1.9 Symbol code file in PINOT	23
4.1.10 Control code file in PINOT	24
4.1.11 Scanner class file in PINOT	25
4.1.12 Parser class file in PINOT	25
5 Refactor plan	26
5.1 File Structure	26
5.2 Refactoring fat files	26
5.2.1 Ast module refactoring plan	27
5.2.2 Control module refactoring plan	28
5.3 Refactoring plan to address fat files	29

6	Refactored Architecture	30
6.1	Applied refactoring	32
6.1.1	File Structure	32
6.1.2	Ast module refactored status	32
6.1.3	Control module refactored status	32
6.2	Analysis of the Refactored PINOT	33
6.2.1	Overall Structure of PINOT	33
6.2.2	Fat Files of PINOT	33
6.2.3	Comparing PINOT Quality Metrics	35
6.3	Evaluating the correctness of the Refactored PINOT	36
6.4	Challenges faced while refactoring PINOT	37
6.4.1	Circular dependencies	37
6.4.2	Discontinued support for Jikes	37
6.4.3	Custom memory allocation	38
6.4.4	Deep inheritance tree	38
6.4.5	Incompatible compiler(s)	38
6.4.6	PINOT vs Modern Java code	38
7	Future Research	40
8	Conclusion	41
	References	42
A	Troubleshooting	43
A.1	Building PINOT	43
A.1.1	Cannot convert 'bool' to 'State*' in initialization	43
A.2	Building .cpa files for Structure101	44
B	Figures	45
B.1	Object-Oriented Metrics Analysis	48
B.1.1	PINOT OOM Analysis	48
B.1.2	Ast OOM Analysis	50
B.1.3	Symbol OOM Analysis	51
B.1.4	Control OOM Analysis	53
C	Relative Comparison	58
D	Version History	59

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 the coherency, maintainability and, quality of the system. Design patterns define best practices for frequently occurring problems in the form of an abstract OO design. A system can easier be analyzed when the applied design patterns indicate the relation between the different objects in a system. Design patterns come especially in handy when analyzing undocumented legacy systems. PINOT(Pattern INference and recOverY Tool) is a pattern recognition tool that can be used to analyze the design patterns used in Java applications^[1].

Design patterns are adopted into a system to solve some recurring problems in an efficient, clean, and reusable fashion. Detecting the presence of these patterns within a system can be beneficial in understanding the unique intent of the code ^{[1][2]}. Without understanding the intent of the code and the design patterns used in an existing system, it is often tedious to perform extended development and maintenance over the code. Legacy software's that have survived the challenge of time have steadily started migrating to modern platforms, and pattern recognition tools like PINOT are priceless entities in this endeavor.

Our goal is to analyze the structure of PINOT and refactor its 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, 3]. Then we will use reverse engineering tools to analyze the structure of PINOT in section 3.

In Section 3 we will first look at how to prepare the PINOT application for analysis. We have analyze 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 Structure101 C/C++ tool¹ 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 Structure101 generic tool³ 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.

Section 4 addresses the various components that were detected during the analysis their intent, and any crucial observations. Important class files that are essential for PINOT are also mentioned in this section along with the Jikes compiler which is the core component of PINOT's functioning. In Section 5 we discuss the refactoring plan that we compiled to be performed on PINOT after understanding the composition of PINOT from manual analysis, and using tools like Structure101 and Understand. In Section 6 we elaborate the refactored architecture of PINOT and compile the quality metrics recorded for the new architecture along with its structural complexity in comparison to the Original PINOT architecture. We also evaluate the refactored PINOT for correctness using reference output generated prior to our refactoring exercise. The potential extension ideas for PINOT are discussed in Section 7 and we conclude the report in Section 8.

¹<https://structure101.com/binaries/structure101-studio-cpa-win64-5.0.16419.exe>

²<https://www.scitools.com/>

³<https://structure101.com/downloads/>

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 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 an 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[4]. The structural design pattern recognition approach is graphically represented in figure 1.

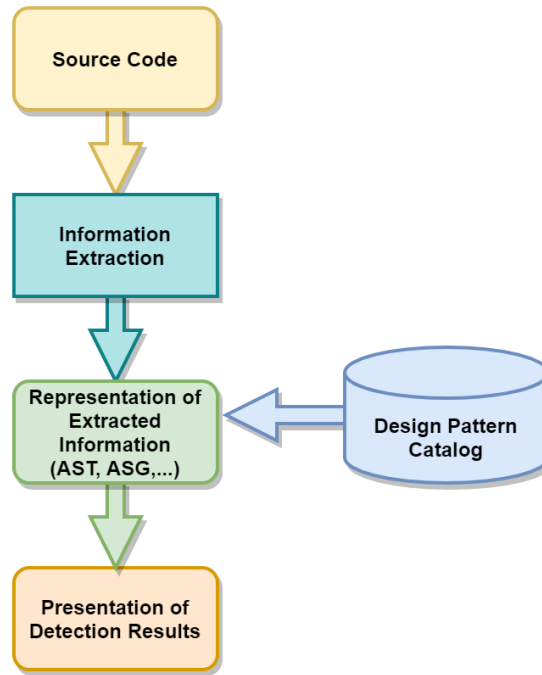


Figure 1: Main steps of structural design pattern detection[4]

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, 5]. 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 patterns which are classified into 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 categorization 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 architecture without directly influencing the system's behavior. 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[3]. PINOT supports the reverse engineering of the structural and behavioral-driven categories as defined in section 2.1. PINOT is founded on Jikes⁴, 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 starts 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 the 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`

⁴<http://jikes.sourceforge.net/>

- 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
```

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.

```
Facade Pattern.
HomeTheaterFacade is a facade class.
Hidden types: PopcornPopper TheaterLights Screen Projector
Facade access types: HomeTheaterTestDrive
File Location: /mnt/d/Academics/pinot/build/bin/designPatterns/HomeTheaterFacade.java

Facade Pattern.
EntertainmentFacade is a facade class.
Hidden types: Nexus Amplifier Projector
Facade access types: EntertainmentFacade
File Location: /mnt/d/Academics/pinot/build/bin/designPatterns/EntertainmentFacade.java

-----
Pattern Instance Statistics:
-----
Creational Patterns
=====
Abstract Factory      7
Factory Method        8
Singleton             5
-----
Structural Patterns
=====
Adapter               3
Bridge                2
Composite             5
Decorator             5
Facade                13
Flyweight             1
Proxy                 10
-----
Behavioral Patterns
=====
Chain of Responsibility 0
Mediator              84
Observer              12
State                 3
Strategy              40
Template Method        1
Visitor               0
-----
Number of classes processed: 442
Number of files processed: 540
Size of DelegationTable: 2012
Size of concrete class nodes: 370
Size of undirected invocation edges: 213

nMediatorFacadeDual/nMediator = 1/84
nImmutable/nFlyweight = 0/1
nFlyweightGoFVersion = 0
```

Figure 2: PINOT output before refactoring.

3 Analysis of PINOT

This section will first explain how the PINOT source code has been analyzed followed by the analysis strategies and outcomes. The reverse engineering application Structure101 has been used to analyze PINOT.

Structure101 originally provided a generic reverse engineering tool that based its C++ analyses on a second analyses tool called Understand⁵. Structure101 just released a new beta version of their tool specifically designed for analyzing C/C++ applications. We used both tools to analyze PINOT and the high over results from both will be analyzed. After these high over overviews, the new Structure101 version for C/C++ will be used to provide more detailed analyses on PINOT.

3.1 Preparing the Source code for Analysis

The application Structure101 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 than initially expected. To compile PINOT, the source code has slightly been modified. These modifications are described in appendix A.1. The Clang compiler directive ast-dump has been used to generate the AST files⁶. The code listing below shows the complete command used to compile PINOT and generate the corresponding AST files.

```
1 clang++ -c
2     -ferror-limit=0
3     -fno-delayed-template-parsing
4     -fno-color-diagnostics
5     -Xclang
6     -ast-dump
7     <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 Structure101 is utilized to translate the AST files into a CPA file that can be used by the Structure101 application to visualize the source project. The bash command used to generate the CPA files are is as follows,

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

Listing 2: Generate CPA from AST dumps.

The building process has been automated using a bash script. The script can be found in our Git repository⁷, the contents of the script are as follows,

```
1 #!/bin/bash
2
3 SRC_DIR="/mnt/d/Academics/SoftwareEvolution/pinot/src"
4 AST_DIR="/mnt/d/Academics/SoftwareEvolution/pinot/src/ast_dump"
5 CPA_FILENAME="Pinotv1.cpa"
6 file_ext="cpp"
7 EXCLUDE_LIST=""
8
9 mkdir "$AST_DIR"
10 mkdir "cap_out"
```

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>


```

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 Analysing PINOT using Structure101 tool

This section describes the architecture of PINOT using Structure101. Firstly the new beta version Structure 101 C/C++ will be used to analyze PINOT. Thereafter the original way of analyzing C++ applications of Structure 101 is used, which utilizes Understand and the generic reverse engineering engine of Structure 101. Finally, the analysis results of Understand will be described as well.

3.2.1 Structure101 C/C++

This section highlights the analysis results of Structure101 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 3.

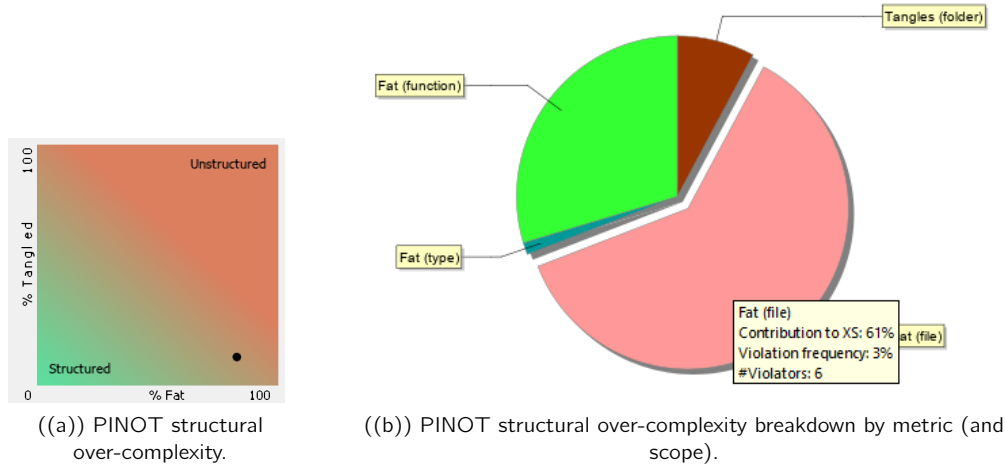


Figure 3: PINOT files by complexity and size.

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 Structure101 generic

Now the highlights in the architecture and design pattern of PINOT will be analyzed using Structure101 generic. Besides describing the analysis results of Structure101 generic, this section will also indicate the differences between these analyses and the analyses of Structure101 C/C++. For that reason this section contains similar graphical representations as section 3.2.1.

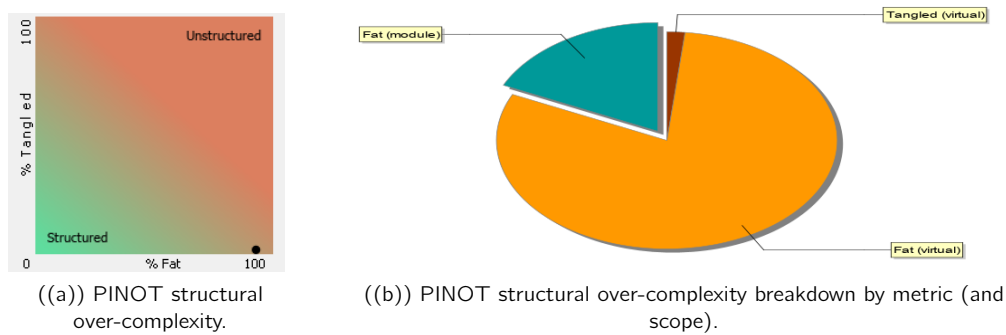


Figure 5: PINOT files by complexity and size.

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 Structure101 C/C++. The results are graphically represented in figure 38

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 Analysing PINOT using Sci-tools Understand

To analyze PINOT with Structure101 generic, the Sci-tools Understand analyses tool has to be used to generate an abstraction DB of PINOT's structure. Understand provides nice analysis insights as well. These insights are described in this section.

The PINOT source code comprises 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 favorable scenario a good source is considered to contain at least 15% of code comment⁸.

⁸<https://everything2.com/title/comment-to-code+ratio>

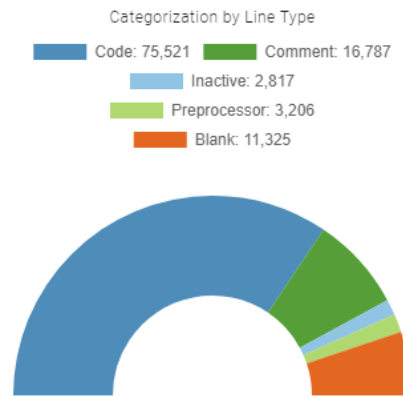


Figure 7: PINOT line breakdown.

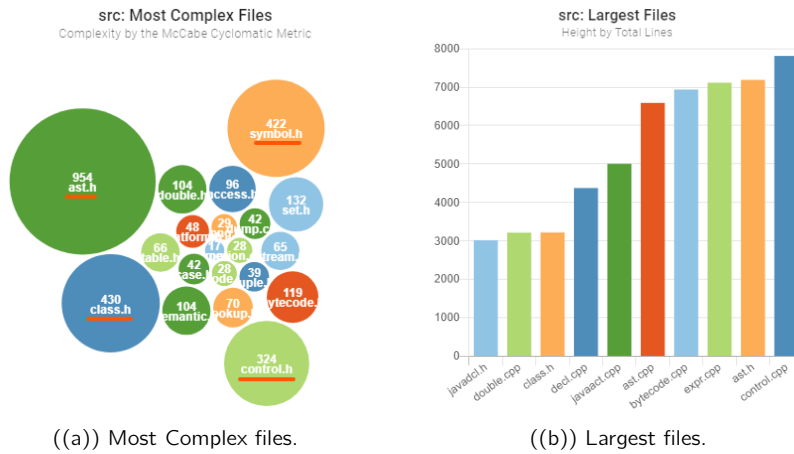


Figure 8: PINOT files by complexity and size.

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.

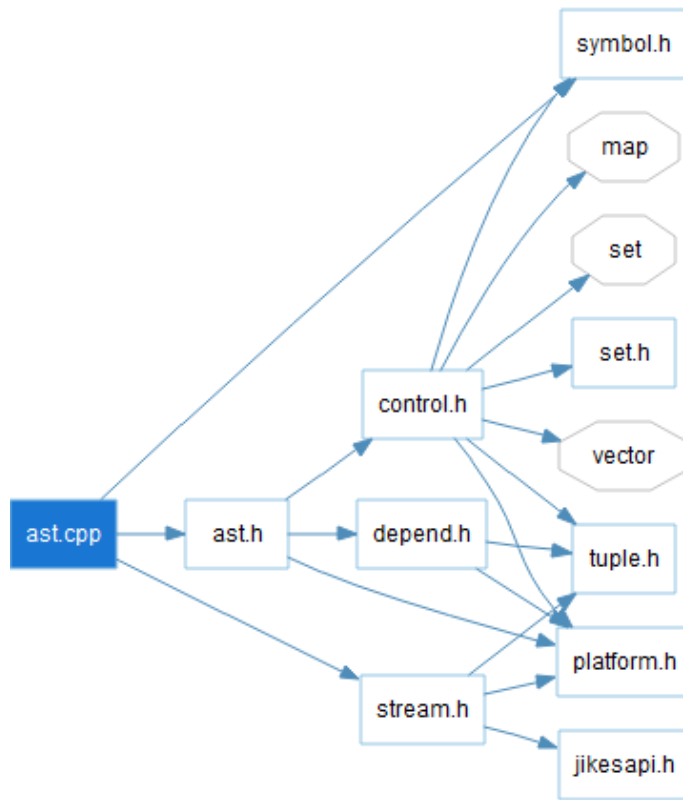


Figure 9: ast.cpp dependency graph.

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.

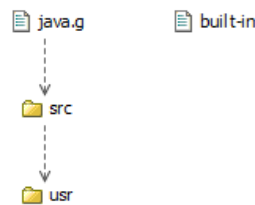


Figure 10: Pinot source code root composition.

From figure 46 in the appendix, 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 guesstimate⁹ that the cluster on the right marked blue relates to the programs entry point, and configurations. Whereas the clusters red, turquoise, and green comprises 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 PINOT's system requirements are clustered in this folder as shown in figure 11.



Figure 11: Structure101 PINOT source code usr folder cluster.

3.4 Architecture Diagram of PINOT

The architecture for Pinot was generated using the Structure101 views. The resulting structured diagram can be viewed in figure 12 in the appendix. The code groups are color-coded as per their groups or clusters. On performing a 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 ones marked in blue serve as the backbone of the project. The implementations generally relate to the data flows, data types, scanners and, tables. These files include the basic structure of Jikes that PINOT was built 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.

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

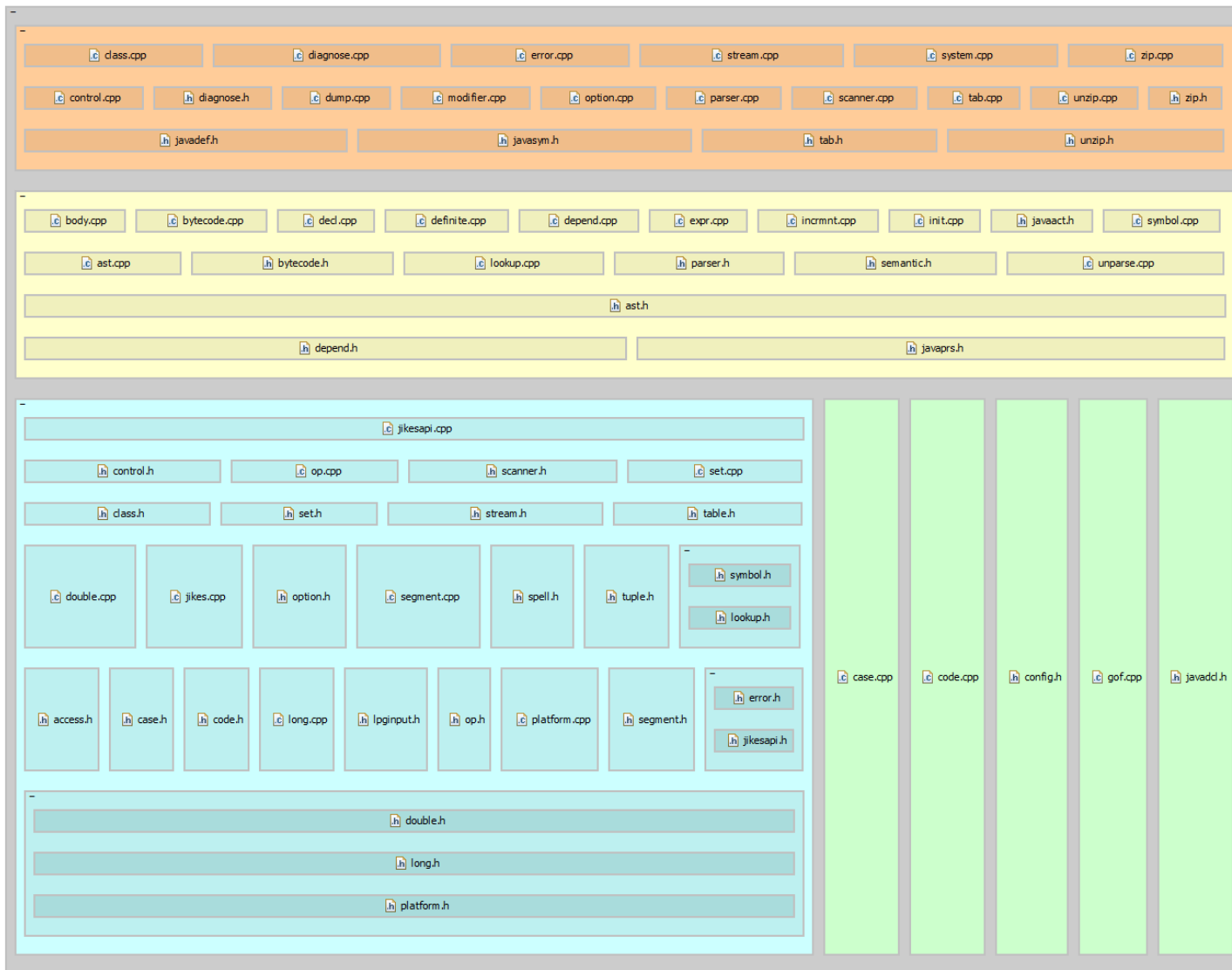


Figure 12: Structure101 PINOT source code Architecture.

3.5 Dependencies

Figure 47 in the appendix, shows the dependency graph that divides the application into 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 consist either of 2 or 3 files.

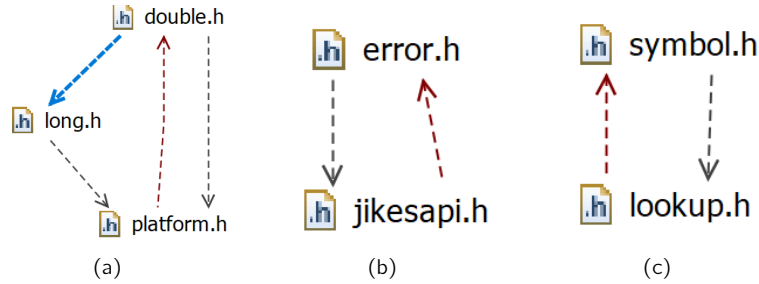


Figure 13: Tangled clusters in PINOT.

The largest tangled cluster consists of the files *double.h*, *long.h*, and *platform.h*. As figure 13 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 respectively *error.h* and *symbol.h*.

3.6 OO-Metrics Analysis of PINOT

The Class OO Metrics Report provides the following object-oriented metrics for each class that is analyzed [6]:

- 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 analyzing the OO Metrics of PINOT as a whole, followed by the individual analysis of the fat files that were found in the Structure101 C/C++ as shown in table 4. The analysis should provide us with insights as to why these files are fat in an Object-oriented design perspective. We will plot a pie chart to understand which OO Metrics has the highest influence and pick the significant ones for a deeper analysis.

3.6.1 PINOT Objected-Oriented Metrics Analysis

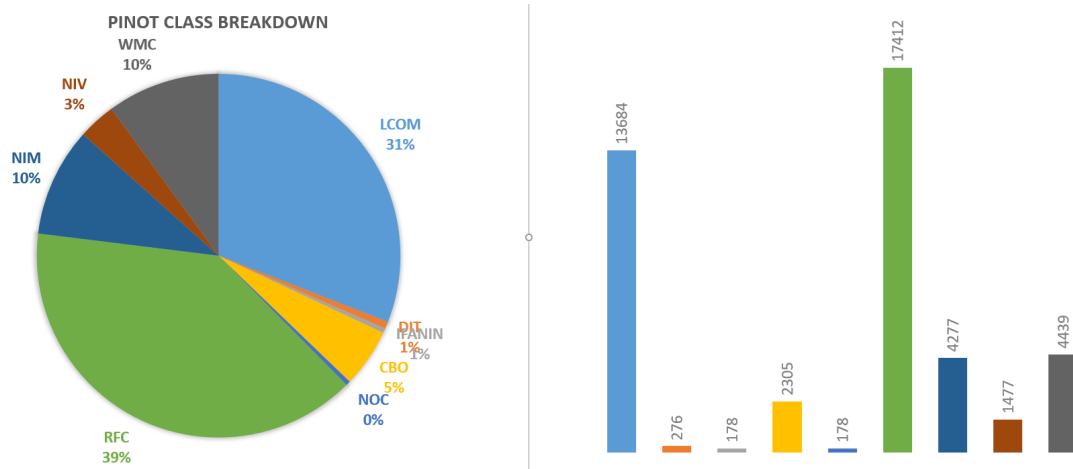


Figure 14: PINOT OO Metrics breakdown.

The figure 14 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 a high count of methods at 39% or 17412 (RFC), and 10% count of Methods (WMC) and Instance methods (NIM) that is already a part of RFC.

Further analyzing 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 49 and 48, which can be found in the appendix. One of our primary goals 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 the appendix and are shown in figure 50 and 51. 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 the same.

3.6.2 Analysing the Ast Class

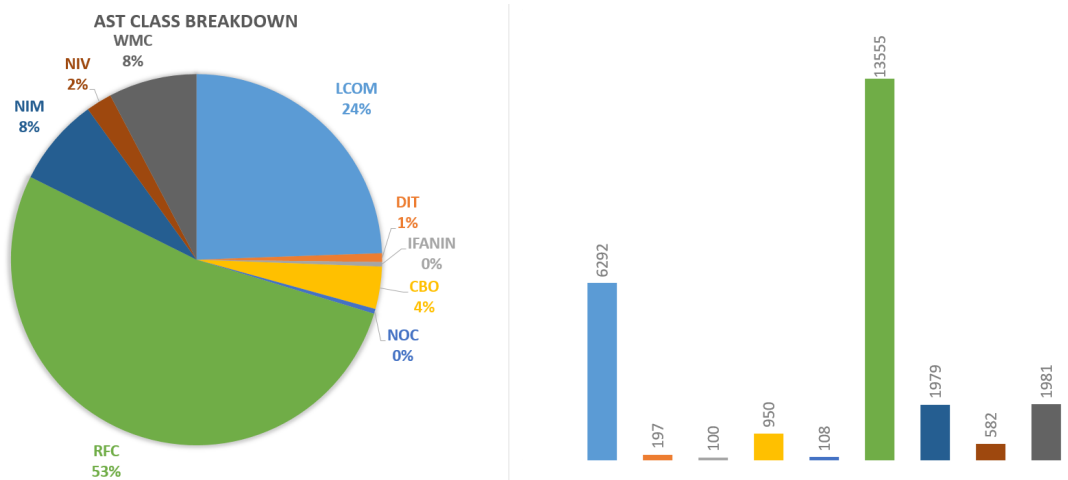


Figure 15: PINOT Ast class OO Metrics breakdown.

From figure 15, 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 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 LCOM and RFC classes for the Ast package can be found in appendix figures 53 and 52. WCM and NIM were left out from deeper analysis as they only contribute towards 8% of the bulk.

3.6.3 Analysing the Symbol Class

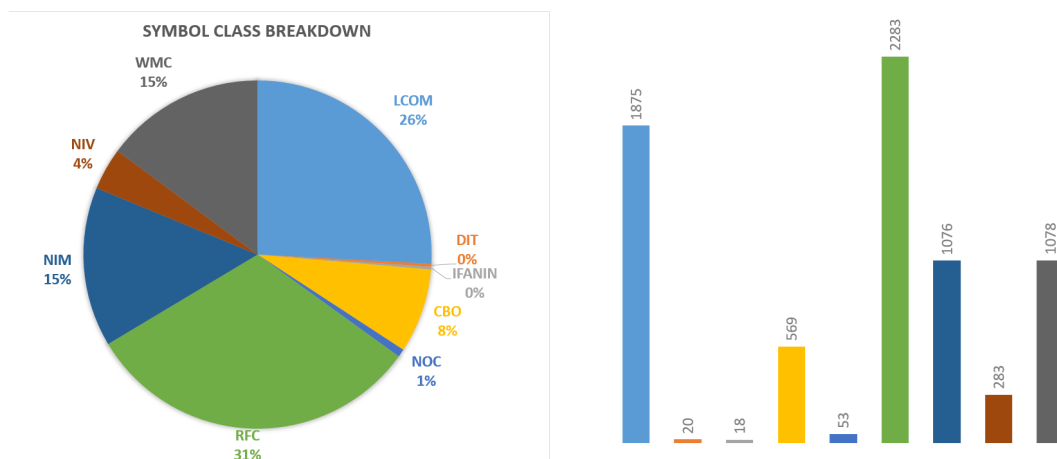


Figure 16: PINOT Symbol package OO Metrics breakdown.

As expected from figure 16, the bulk resides in the LCOM (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 55, 54, 57 and 56. 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 themselves. 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 the Ast package would passively reduce the fatness of the Symbol class.

3.7 Analysing the Control Class

The significant portion of the OO Metrics is again comprised of the LCOM (27%) and RFC (34%), some minor contribution from the WMC (13%) and NIM (13%) as shown in figure 17.

The top 10 LCOM, RFC, WMC, and NIM can be found in figures 59, 58, 61 and 60, which are listed in the appendix. On observation, we see that the class **Semantics** has a major contribution towards the fatness of the Control class as it amounts to a significant amount towards the RFC, WMC and, NIM metrics. The LCOM 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.

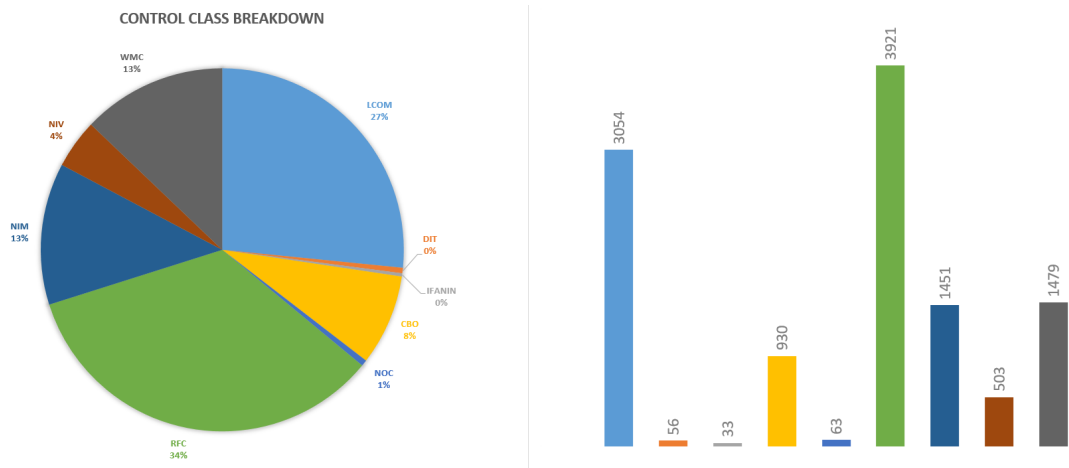


Figure 17: PINOT Control package OO Metrics breakdown.

4 Component Analysis of PINOT

In this section, we will discuss the role of the important components in PINOT and also see why some of these components are fat.

4.1 Understanding the PINOT source

The PINOT source code is clustered into 6 clusters and some unclustered files. For this analysis, we will first start by understanding the core platform information of PINOT.

4.1.1 Declaration module

Figure 18 shows the PINOT cluster which mainly comprises the header files. It is a good practice to only write declaration onto the header files, but PINOT adopts a different approach and has added its core implementations as headers.

Important dependencies like the data types and structure (long, double, tuple, segment, set and table) are declared and also sometimes implemented at this scope if required. **op.h** and **op.cpp** have custom operators (example, "iconst_0", "push int constant 0", 0,) and options that is internally defined and used by the PINOT application.

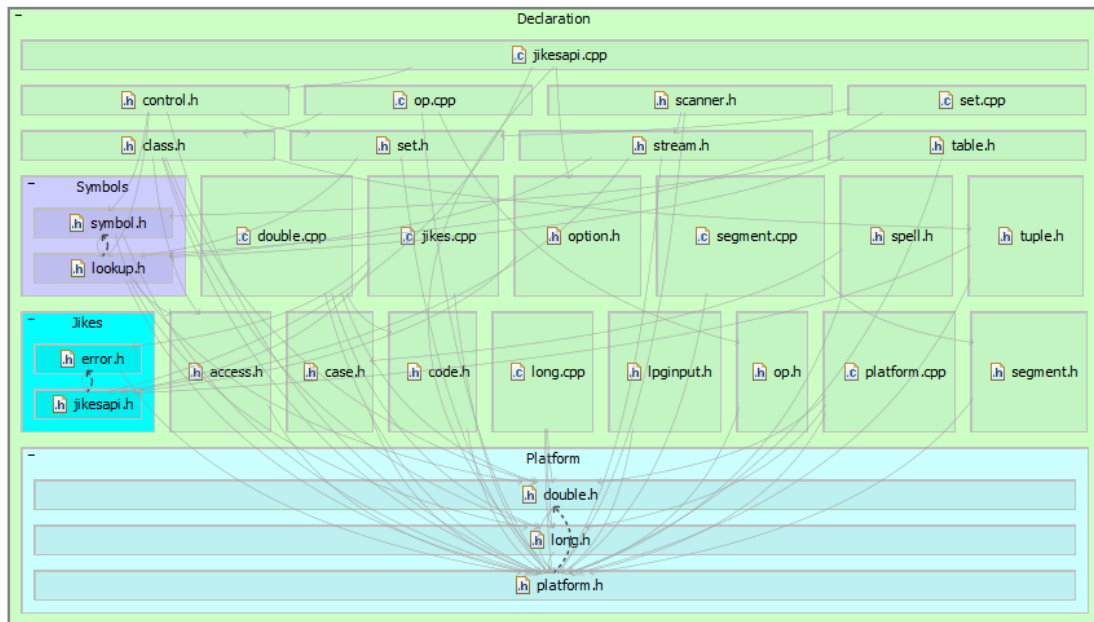


Figure 18: PINOT Declaration module.

4.1.2 Generation of Abstract Syntax Tree

Figure 19 shows the cluster responsible for housing the PINOT's version of abstract syntax tree. `ast.h` is the primary class that contains most of the classes that relate to the Abstract syntax tree. The objects parsed by jikes using the java grammar will be translated into one of the equivalent classes within this cluster. In an essence, we can view this layer as the domain-specific language of PINOT that will be used to read and analyze the AST of the input java file.

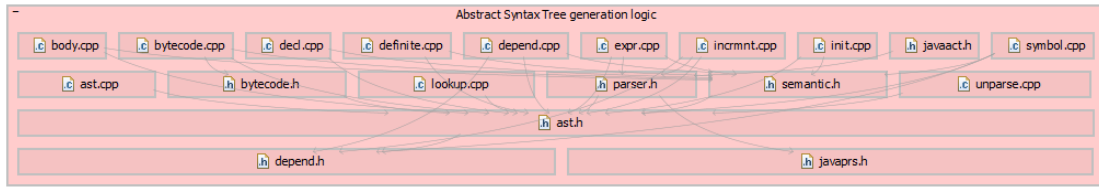


Figure 19: PINOT AST generation module.

4.1.3 Orchestration and AST analysis

The orchestration layer is responsible for controlling the execution patterns of PINOT. There are no specific controller patterns adopted by PINOT to achieve this, but simple switch cases are utilized to navigate the execution branches. The layer also reads the Ast objects generated in the AstGeneration layer, analyzes the syntax tree and, aggregates the results to be displayed.

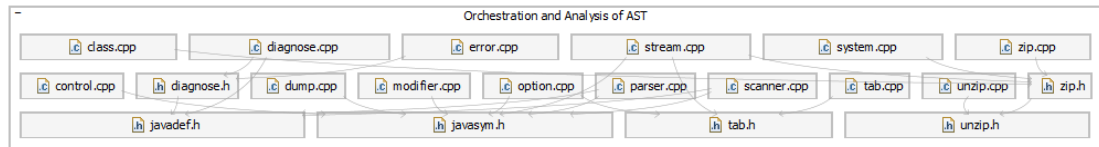


Figure 20: PINOT Orchestration and AST analysis module.

4.1.4 Standalone classes or Unclustered classes

todo: case.cpp and code.cpp was moved; update the image The classes do not fall under a particular cluster. Like the config.h has a transitive dependency through platform.h to all the clusters and on the contrary the gof.cpp file is unused by either of the clusters.

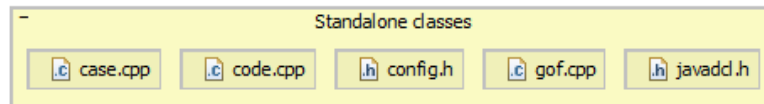


Figure 21: PINOT Standalone class group.

4.1.5 Platform cluster

The platform cluster is mainly responsible for setting up the PINOT environment dynamically based on the host machine's software availability. This is predominantly handled by the **platform.h** file. An example of this is shown below, where the code checks for the availability of windows.h file and if it exists the file is included in PINOT's environment.

```
#ifdef HAVE_WINDOWS_H
    #include <windows.h>
#endif
```

Other files that require the environment would simply need to import just the **platform.h** file. The presence or absence of these required files was being aggregated while running the **configure** command during the installation of PINOT, as shown in Section 2.2.1.

double.h and long.h: These are the custom implementation of the double and long data type. The implementation(s) include the redefinition of arithmetic operators, logical operators, comparison operations and, some additional utility functions.

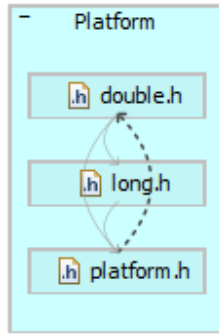


Figure 22: PINOT Platform cluster.

4.1.6 Jikes Compiler

Jikes is a Java compiler written in C++, as of 2010 update for the said technology has been discontinued [7]. PINOT utilizes Jikes to compile the input java files that were supplied by the user for GoF pattern analysis. This also ensures that the Java compiler and PINOT source code have the same programming platform. The **jikesapi.h** is a class with states that can be used to configure the Jikes compiler. The **error.h** class is a specific implementation to fix an issue where **basic_ostringstream<wchar_t>** was not supported by all the compilers.

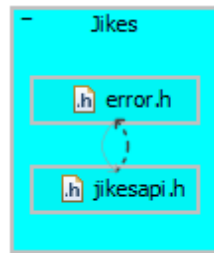


Figure 23: PINOT Jikes module.

Java grammar: Jikes compiler utilizes java grammar to compile a java source code. A java grammar is a file that contains a set of rules that will be consumed by the jikes compiler to parse the java source code into a respective C++ object defined in the rule, figure 24 shows a sample rule that is used to assign java integers as IntegerLiterals. The flow of how PINOT reads and analyses a java source code is also depicted in the figure. The input is read by the Jikes compiler, which later parses the file(s) based on the Java grammar specified to it, the parsed objects are in PINOT understandable object types which are later used by PINOT for analysis.

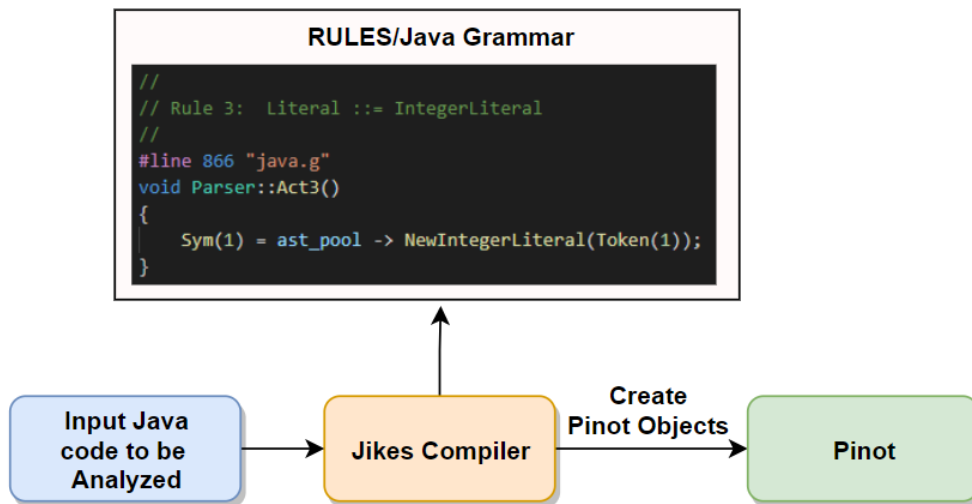


Figure 24: PINOT Jikes rules and Java grammar.

4.1.7 Symbols cluster

The **Symbol.h** is an important class implementing the various SymbolTable like data-structure classes that will be used by PINOT to store and retrieve attributes. **TypeSymbol** is one such domain-specific symbol implementation where it keeps track of a class, its nested classes if it is a base class and, so on. These data structures are used similar to the building blocks of a domain-specific language for PINOT and they all extend a base class called Symbol in the **lookup.h** file. The Symbol class consists of member variables like name, package, method and so on that can be used to express the components of a class file that is under analysis.

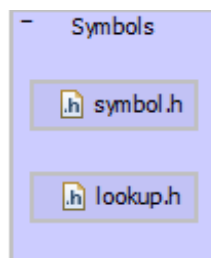


Figure 25: PINOT Symbol module.

4.1.8 Ast code file in PINOT

Figure 26 shows the dependencies and transitive dependencies of the Ast.h header file. The file is also the one that contributes the most towards the fatness of the PINOT source code.

Role: This file contains the definitions of the classes used to construct the AST (Abstract Syntax Tree) representation of a Java program. **class Ast** functions as the base case on which the other Ast node classes are defined on. The classes directly translate as the grammar required to create an Ast tree, some of classes with this file are named as **AstTrueLiteral**, **AstFalseLiteral**, **AstStringLiteral** and so on. Simply put, each data type parsed is admitted as tokens and later initialized to an appropriate sub-class of the Ast class type. The initialization is mainly performed using inline method calls. This class is without a doubt the heart of the PINOT source code.

Observations: On analysis of the logic within the Ast class file we found that most of the classes have cyclic dependencies. Also, the **StoragePool** was a custom-built memory allocator class (similar to Malloc) that initialized these Ast classes. The Storage class accounted for 16% of the fatness of Ast class with respect to the lines of code. Refactoring the StoragePool will result in cyclic dependencies as the Ast class requires the StoragePool to initialize Ast nodes and arrays, whereas the StoragePool requires the Ast sub-classes for its inline functions.

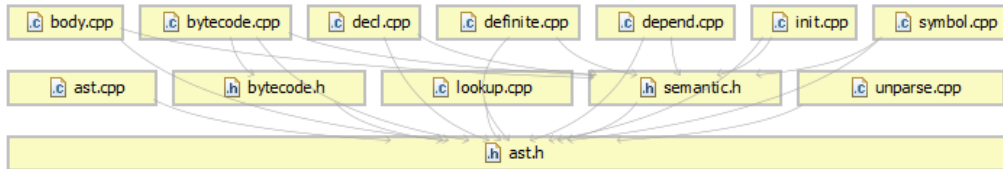


Figure 26: PINOT Ast dependencies.

4.1.9 Symbol code file in PINOT

Figure 27 shows the various dependencies and transitive dependencies of the Symbol implementation. The class is the second class that contributes towards the fatness of PINOT.

Role: If Ast functions as the grammar that translates the Java code into PINOT understood language. The Symbol class is responsible for understanding the Java class itself. For example, a **class MethodSymbol** is responsible for extracting if a java class has accessible private variables, the methods return type, its constructor and, so on. The class has a strong dependency with the Ast class, as it utilizes and/or understands the Ast sub-classes to perform its operations.

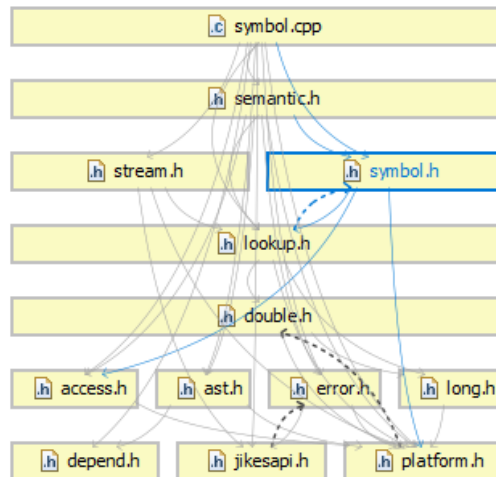


Figure 27: PINOT Symbol dependencies.

Observations: The class contains some classes like **FileLocation** that are not its sub-class but

can be looked at as utility classes, these classes can be moved to separate utility class. The class contains fairly fewer inline functions but has a cyclic dependency with the Ast class which should be carefully accounted for during the refactoring.

4.1.10 Control code file in PINOT

Figure 29 shows the dependencies and transitive dependencies of the Control class definition and implementation.

Role: It is the orchestration class that invokes the Symbol class functionalities to obtain the patterns. The class contains classes like **SingletonAnalysis** (checks if a parsed java class is Singleton using variable and method symbols), **FactoryAnalysis** that looks for factory pattern and so on.

A control flow is an order in which a set of statements, instructions or, function calls of a program are invoked to achieve the required state of the system [8]. The code snippet responsible for the **control flow** of the PINOT can be found in the **control.cpp** file starting from line:6056 (original) and line:4852 (refactored). The snippet starts of by extensively checking for any bad tokens, storage, tables and, so on, and later invokes a series of function calls to find the GoF patterns as shown in Figure 28.

Observations: The class contains a Utility class within it, it should be moved to a common header for reuse. The main class responsible for orchestration is **class Control** which also provides cache support for some of the objects, the rest of the Analysis orchestration can be moved to their respective classes based on the design pattern being looked at, it can passively promote readability.

```
Output << "----- Original GoF Patterns -----" << endl << endl;

//FindSingleton(cs_table, ms_table);
FindSingleton1(cs_table, ast_pool);

FindChainOfResponsibility(cs_table, ms_table, d_table, ast_pool);
FindBridge(cs_table, d_table);
FindStrategy1(cs_table, d_table, w_table, r_table, ms_table);
//FindFlyweight(mb_table, gen_table, assoc_table);
FindFlyweight1(ms_table);
FindFlyweight2(cs_table, w_table, r_table);
FindComposite(cs_table, d_table);
//FindMediator(cs_table, d_table);
FindTemplateMethod(d_table);
FindFactory(cs_table, ms_table, ast_pool);
FindVisitor(cs_table, ms_table);
FindObserver(cs_table, d_table);
FindMediator2(cs_table);
FindProxy(cs_table, d_table);
FindAdapter(cs_table);
FindFacade(cs_table);
```

Figure 28: PINOT Control flow invocations to find GoF patterns.

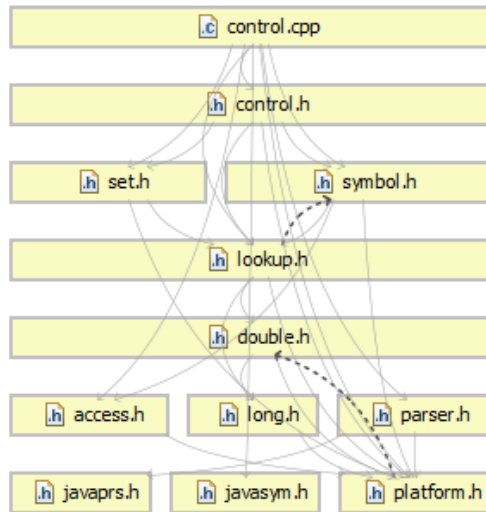


Figure 29: PINOT Control dependencies.

4.1.11 Scanner class file in PINOT

The scanner class is a series of if and else conditions that detect the tokens sent by jikes using the stream.h class file. The stream class file encapsulates details related to reading a stream of possible encoded data from the file system. The LexStream type is utilized to construct this token stream, where a newline character indicates the start of a new token and the stream always terminates with a carriage return. The scanner is equipped to classify character literals, string literals, integer literals, some special characters, keywords and, Id's.

The scanner class is referenced by the **option.h** file and transitively the **option.cpp** file. The option module is used to process a set of external arguments that can be fed to the PINOT during its execution, this is similar to how we can modify the nature of Java compilation/execution using command line arguments. These options provided are treated as tokens and are processed by the scanner.h and, scanner.cpp code files.

The scanner class does utilize some methods from the jikesapi like reportError() to log any error during its scan and some variables that are used to check if the compatible SDK version is being used. The tokens that can be processed by the scanner file can be found in javasym.h file and the jikes enumerations used to set the state in javadef.h files.

4.1.12 Parser class file in PINOT

The parser class is a utility class that is utilized by the Ast.h file for parsing its object types. Package header is the fully qualified package name present in a java class file, the Ast.h type might simply store it as an object but the logic used to navigate through this object, check for errors and extract required information is present in the parser class file. Similarly to this, stack operations are defined to facilitate ListNode operations and overloaded methods are provided to parse different variants of AstClassBody types.

5 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. With the help of Structure101 and by studying the source code a refactor plan has been designed. This section will introduce that refactor plan for PINOT which is focused on improving the file structure and decreasing the file size and thus fatness of PINOT.

5.1 File Structure

Currently, the file structure of PINOT is completely unstructured because all the source files are grouped in one single folder. The file structure is the easiest way to indicate a little structure in a software project. While PINOT does not contain any file structure yet, from the Structure101 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-folders to create a logical separation of source files based on functionality. Moreover, the auto `levelizer` function in Structure101 performs precisely that restructuring of the source files.

1. **Partition files to sub-directories** - The base file structure that will be created can be deduced from the dependency graph that is displayed in figure 47. 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. Figure 62 in the appendix shows the refactoring plan to address the fat folders issue in PINOT, the structural complexity estimated on the application of this refactoring steps is as shown in appendix figure 63 with 48% fatness and 17% tangle, we estimate that there would be a small increase in tangle which is an acceptable trade-off in comparison to the fatness reduced. The refactoring steps are as follows,
 - Move the files to the 3 clusters declarations, Ast generation and, Orchestration as shown in the appendix figure 62.
 - Move `pinot/src/ast.h/Parser` to `src/AST_generation/ast_parser/Parser`
 - Move `pinot/src/ast.h/StoragePool` to `src/AST_generation/storage/StoragePool`

5.2 Refactoring fat files

In this section, we will address the fat files in PINOT and the strategies adopted to refactor these classes. On analysis of PINOT source code, we determined that the fatness of these fat files are not due to the presence of a single large classes but many small classes are clustered into a single folder. In C++ a `#include <header>` is equivalent to copying the contents of header files into a different file, so splitting the header files alone will not reduce the fatness of PINOT, a collaborated movement of both declarations and implementations will provide better results.

5.2.1 Ast module refactoring plan

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*. The `StoragePool` class consists of approximately 1000 lines of code. Moving these lines to its own file greatly reduces the fatness of *ast.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.

5.2.2 Control module refactoring plan

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 analyzing 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 Variables and Methods** - The control implementation file contains global variables to count the occurrences of patterns in the to be analyzed 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. For example *FindSingleton(..)* is superseded by *FindSingleton1(..)*.
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.
5. **Analyses Classes** - The control files define 6 different analyses classes to support the analyses; *ChainAnalysis*, *ControlAnalysis*, *CreationAnalyses*, *FactoryAnalyses*, *FlyweightAnalyses*, and *SingletonAnalysis*. These classes will all be moved to their own header and implementation files to decrease the fatness of the control files.

5.3 Refactoring plan to address fat files

Figure 64 in the appendix shows the refactoring plan crafted in Structure101 to break down the fat files of PINOT into logical splits. It was observed that the Ast.h class mainly comprised of 3 types of classes Statements, Literals and, Expressions. A decision was taken to move these classes to the core library. This refactoring not only reduces the fatness of ast file but also makes it easy for the end-user to navigate through the different statements, literals, and expressions supported by PINOT. The control file contains classes that define the GoF patterns, these classes were moved to a patterns folder for clear distinction from orchestration (control file). The figure 65 in the appendix shows the estimated structural complexity of PINOT (47% fat and 31% tangle) after applying these refactoring steps, the tangle is again slightly increased but the fatness is also slightly reduced. The small reduction might appear negligible, but the ast class files are comprised of many small classes, in comparison to the rest of the classes the ones moved provided a better reduction in fatness. During our analysis we also noticed that moving some files caused a very sharp increase in tangle value up-to 90%, thus care must be taken while breaking these class files. The refactoring strategy also considers the user's readability into consideration to provide a sense of logical grouping to PINOT's classes.

Figures 30 and 64 shows the expected structural complexity and the fat-files with their respective values after executing the fat files refactoring plan on the PINOT source code. We see that the codebase is still fat this is because that PINOT comprises of numerous small classes that are defined in few files and this refactoring plan has been laid out to address only a select number of classes based on our time estimations. Comparing this to the original values in figure 4, we can observe that the fatness of ast.h shows a 25% decrease in comparison to the original, and a slight reduction of fatness in the ast.cpp files as well. The ast.h file houses the implementation part of the code and thus can be moved effectively in comparison to the ast.cpp file. From the figure 30(a) we also observe that the refactoring plan is introducing small amounts of tangles.

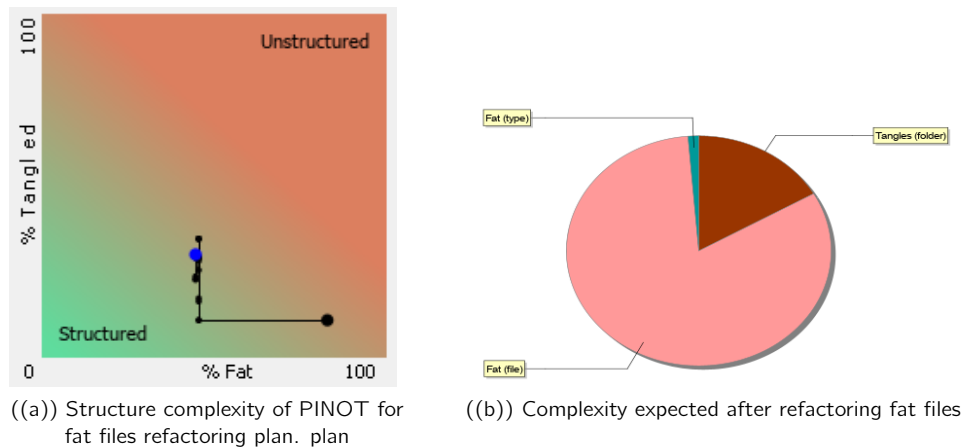


Figure 30: Complexity and Fat-files expected on applying the fat file refactoring plan.

Item	Value
src/AST_generation/ast.h	615
src/AST_generation/ast.cpp	391
src/declarations/symbol/symbol.h	196
src/orchestration/control.cpp	164
src/AST_generation/unparse.cpp	162

Figure 31: Complexity and Fat-files expected on applying the fat file refactoring plan.

6 Refactored Architecture

The application PINOT has been refactored based on the refactor plan described in section 5. This section will describe the architecture of our refactored version of PINOT. First, this section will describe which points in the refactor plan have and which points have not been applied in the refactoring phase. Thereafter the new architecture will be analyzed with the help of Structure101 C++ and Understand. Figure 32 shows the refactored architecture of PINOT after applying the refactoring plan.

The architecture shows the clear distinction of the files into the three categories orchestration, astGeneration and, declaration, this structure was constructed using the auto-levelize functionality of Structure101 that partitions the files into folders or categories based on their similarities of role in the source code.

The orchestration layer mainly comprises of code that is relevant to how PINOT should initiate a chain of calls to perform analysis on the parsed Java input file. The astGeneration layer is where the framework implementation of the PINOT lies, in our refactored architecture we have abstracted the three primary types of classes that build the entirety of the Ast object types that PINOT understands i.e. expression, literals and, statements. These three components are refactored as the **core** module of PINOT. The storagepool is also abstracted out from the ast file that not only accounted for the largest fatness contribution but also restores the single responsibility principal for these classes.

The declaration layer is where most of the header and the backbone implementation is defined. The implementations defined here are mainly used to facilitate the astGeneration layer. Jikes being an important module who's sole purpose is the parse the java input files into PINOT understandable format is moved to a **jikes** folder. GoF pattern classes are found in the control.h files were moved to their separate classes in **patterns** folder. Declarations that were more focused on providing utility and displayed possible code reuse were moved to the **Utility** folders.

A header file should ideally contain the declaration of classes, structs, methods, etc and their implementation should be defined in the respective .cpp files, but PINOT breaks this norm by defining implementation in their header files, this poses a challenge during refactoring as certain implementations like inline methods result in incomplete classes errors.

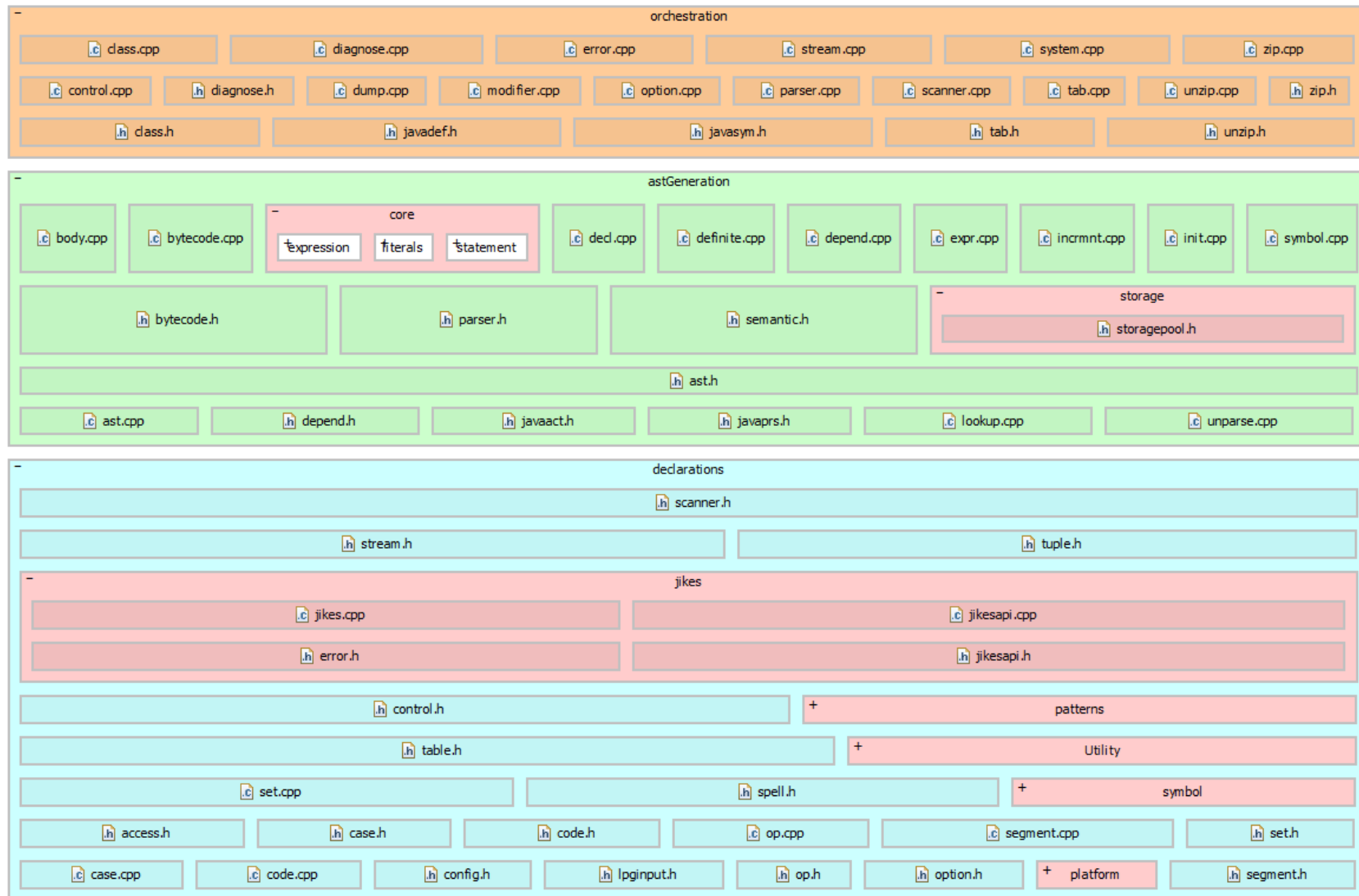


Figure 32: Refactored Architecture of PINOT

6.1 Applied refactoring

The refactor plan that was introduced in section 5 consists of 10 points. For each of these 10 points the list below will describe if it was applied in the refactor phase and why (not).

6.1.1 File Structure

1. Partition files to sub-directories [**Applied**] - The source files of PINOT have been restructured into 5 folders eventually. 3 of these folders, `astGeneration`, `declarations`, and `orchestration` are constructed as originally described in the refactor plan. The other 2 folders, `patterns` and `Utility`, contain files that were created for refactoring points **4** and **5** in Section 6.1.3.

Refactoring plan: R1

6.1.2 Ast module refactored status

1. Unrelated Class[**Applied**] - The class `StoragePool` has been moved to its own header file.
2. Related Classes[**Partially Applied**] - Logically isolated set of classes and functions were detected and moved to their respective headers and class files.

Refactoring plan: R3

3. Extracted Class Methods[**Partially Applied**] - Methods that were common between multiple classes were extracted to a separate utility folder.

Refactoring plan: R4

4. Import Consistency[**Applied**] - All header files that relied on `StoragePool` now include the new header file `storagepool.h`.

Refactoring plan: R5

6.1.3 Control module refactored status

1. Control Constructor [**Deferred**] - During the refactoring we learned that the methods responsible for the pattern mining were not part of a class. These methods were defined in the namespace `Jikes` similar to the global count variables in point **7**. Due to this new information we decided that it would be unwise to simply move the logic to a helper function of the `Control` class. More research has to be done into these methods before they can be refactored.

Refactoring plan: R6

2. Global Count Variables [**Deferred**] - Due to the new uncovered information that is mentioned at the previous point(**1**), we decided to skip this point as well. We think it would be wise to include this point into the research of point **1** as well.

Refactoring plan: R7

3. Remove Unused Methods [**Deferred**] - similar to point **2**, this point is skipped due to the newly uncovered information in point **1** as well. It turned out that the unused methods were part of the global analyses methods. We think it would be wise to include this point in the research of point **1** as well.

Refactoring plan: R8

4. Table Classes[**Applied**] - All the table classes that are mentioned in the original refactor plan have been moved to a new header file. On top of this also related utility classes have been moved to separate header files as well. All these header files are grouped in the new folder `Utility`.

Refactoring plan: R9

5. Analyses Classes[**Applied**] - All the analysis classes that are mentioned in the original refactor plan have been moved to new header files. Also their implementation methods that were located in `control.cpp` are moved to separate implementation files. All these source files are grouped in the new folder `patterns`.

Refactoring plan: R10

6.2 Analysis of the Refactored PINOT

The refactored architecture of PINOT will be analyzed in a similar fashion as the original architecture of PINOT was. Since the original analyses already contain in-depth interpretations and elaborate explanations of the analysis results, the analysis result of the refactored architecture will just be compared against the original results. Naturally, the observed differences in the analysis results will be considered in detail. Both Structure101 and Scitools Understand are used to derive insights. First, the overall complexity will be compared. Thereafter the complexity of the abstract syntax tree and control files will be compared.

6.2.1 Overall Structure of PINOT

The main focus of the refactor plan that has been introduced in section 5, was to reduce the overall fatness of PINOT. The file structure had the **largest influence** on the overall fatness because in the original complexity pie approximately three-quarters of the total fatness was due to **fat folders**. In figure 34, both the complexity pie of the original and the refactored architecture are shown. As indicated by figure 34(b), the complexity due to fat folders has been **removed completely**.

Though the improvement of the file structure as per point R1 of the refactor plan has a large influence on this drop. Also, the improvements made to the fatness of the abstract syntax tree and control files, which will be discussed in more detail later, attributed to the drop in overall fatness. These refactoring attribute to an overall **reduction of fatness** in PINOT by approximately **50%** as indicated by the comparison in figure 33.

While the fatness was reduced tremendously the **tangles** in the refactored architecture **increased** as indicated by figure 33. This increase in tangles can also be observed from figure 34, where a large percentage of the complexity is assigned to tangled folders. The cause for such tangle will be discussed later in section 6.4. Fat folders now contribute to 36% where this used to be only 2%. Of course, this absolute difference does not provide a good comparison due to the complete absence of fat folders in the complexity of the new architecture.

From appendix C, the relative comparison 36% against 7% can be deduced. Although this improves the tangle increase ratio and a small increase was expected beforehand, it still is a remarkable and regrettable increase in tangle complexity.

6.2.2 Fat Files of PINOT

The refactoring of fat files in PINOT is an extension step after the fat folder reduction. In this step, the independent logical components are abstracted from the fat files and moved to new header and class files. The fat files were detected in the analysis phase as described in Section 3 and the refactoring plan laid out to reduce their fatness was discussed in Section 5. The main focus herein was on the files `ast.h`, `ast.cpp`, `control.h`, and `control.cpp`

Comparing figures 4 and 35, we see that the fatness of `ast.h` and `ast.cpp` was indeed reduced from the original values of 814 and 437. But the values obtained after refactoring are slightly higher for `ast.h` than the expected values found while constructing the refactoring plan. Also the fatness score of `control.cpp` dropped with approximately 30 points. Thus the fatness of all files has been

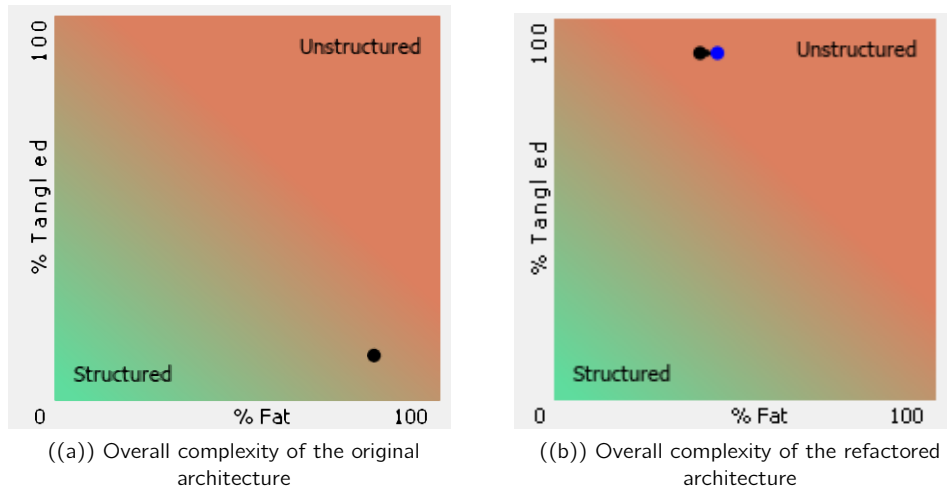


Figure 33: The overall complexity charts generated by Structure101 for the complexity of both the original and refactored architecture of PINOT

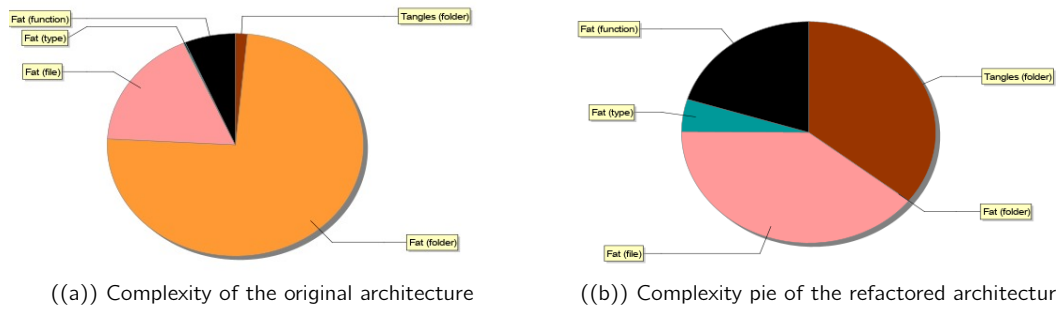


Figure 34: The complexity pies generated by Structure101 for both the original and refactored architecture of PINOT

reduced though there is a negligible difference between the expected results and the obtained results.

Item	Value
src/astGeneration/ast.h	639
src/astGeneration/ast.cpp	390
src/declarations/symbol/symbol.h	196
src/astGeneration/unparse.cpp	162
src/orchestration/control.cpp	132

Figure 35: PINOT fat files after refactoring the fat-files.

6.2.3 Comparing PINOT Quality Metrics

The class breakdown and quality metrics of PINOT that was generated using sci-tools Understand is as shown in figures 36 and 37. While our observations under Structure101 provided us with positive outcomes in regards to the decrease in the fatness of the PINOT codebase. The quality metrics of the refactored architecture is slightly degraded in comparison to the original. From the figures, we can deduce that the original architecture has a 48% lack of cohesion and the refactored architecture is at 60% lack of cohesion which is a 12% increase. This is a fair trade-off for a massive decrease of fatness by 50%.

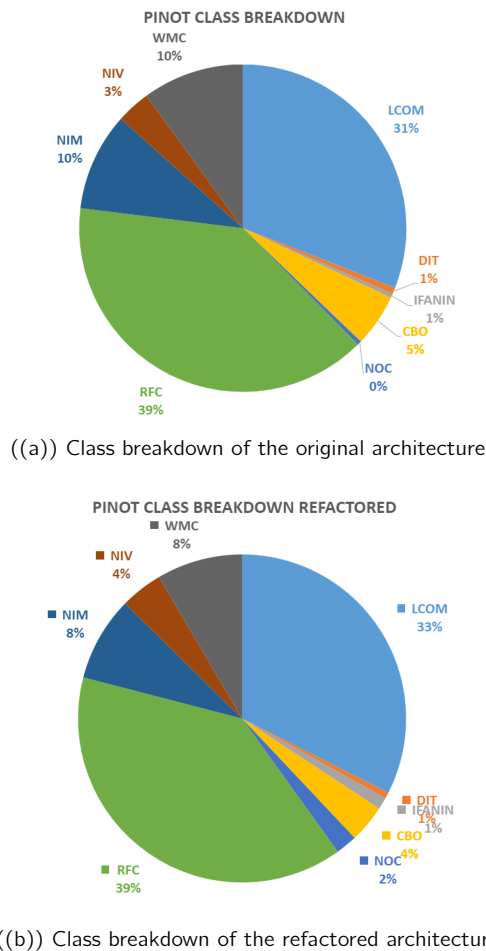


Figure 36: The Class breakdown generated by Understand for both the original and refactored architecture of PINOT

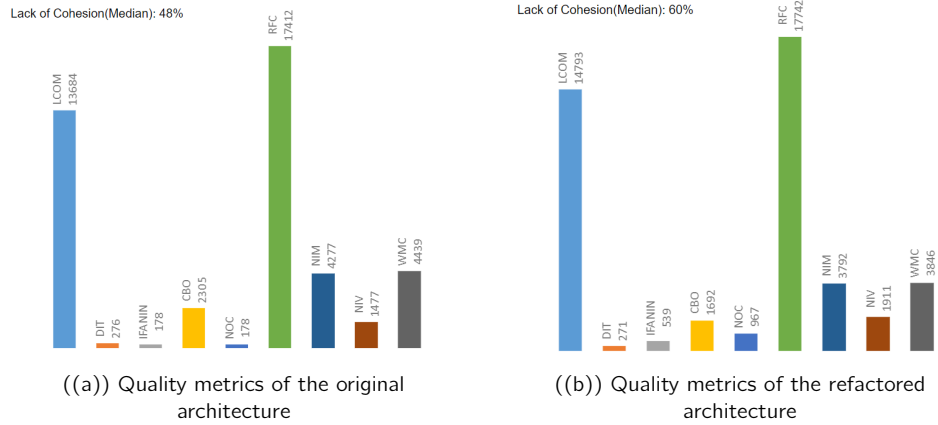
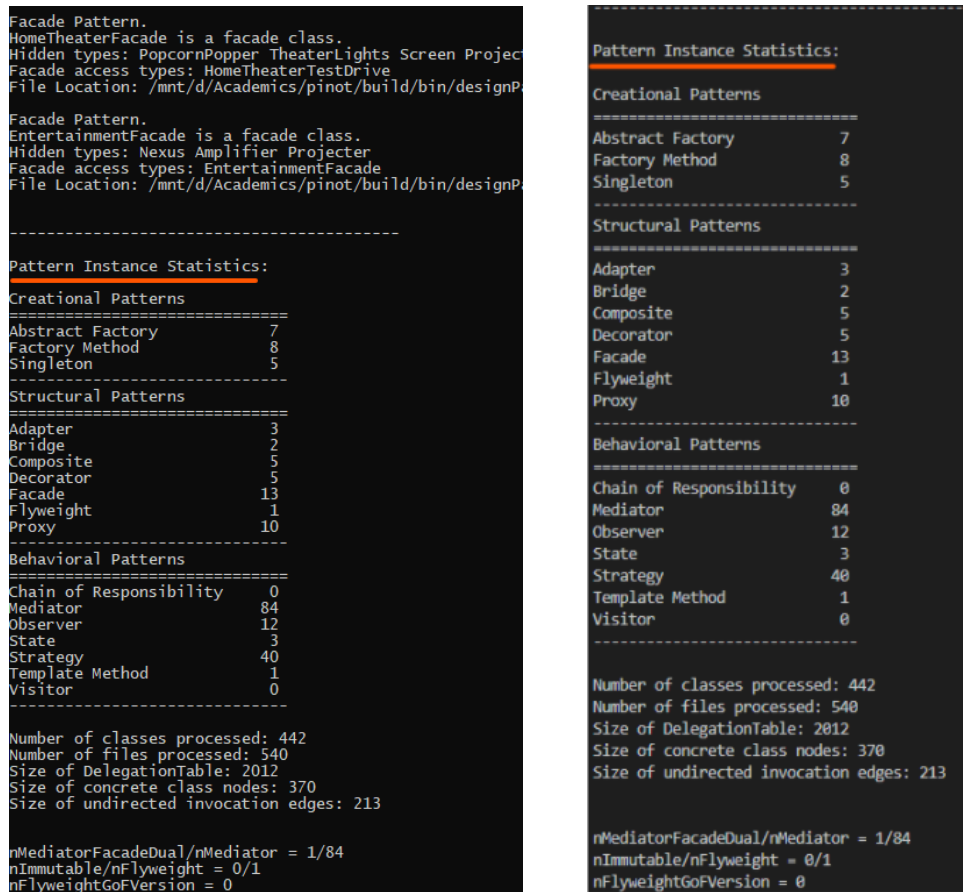


Figure 37: The OO-Metrics generated by Understand for both the original and refactored architecture of PINOT

6.3 Evaluating the correctness of the Refactored PINOT



((a)) PINOT output report before refactoring (expected). ((b)) PINOT refactored output report (actual).

Figure 38: Comparison of PINOT output pre-refactoring and post-prefactoring.

To evaluate the correctness of the refactored PINOT, we analyzed the reference GoF patterns repository used to generate the pre-refactoring output reference, shown in figure ??(a) and compared it against the output report generated by the refactored PINOT, shown in figure ??(b). From the two figures, we see that there are no discrepancies between the two outputs, indicating that our refactoring exercise did not deviate from the original functionalities of PINOT.

6.4 Challenges faced while refactoring PINOT

The section will elaborate the challenges that were encountered while refactoring PINOT.

6.4.1 Circular dependencies

In software engineering a circular dependency is a state where two or more modules directly or indirectly refer to each other [9]. The codebase of PINOT contains multiple instances of coding patterns that are prone to circular dependency issues. Figure 39 represents one of such instances that occurred after refactoring. In this example, the StoragePool class allocate memory locations to the classes present within the ast.h file, and the classes defined within the ast.h file require StoragePool during initialization. StoragePool was the fattest classes within the ast.h file, and moving this file outside revealed the circular dependency that is depicted in the image below.

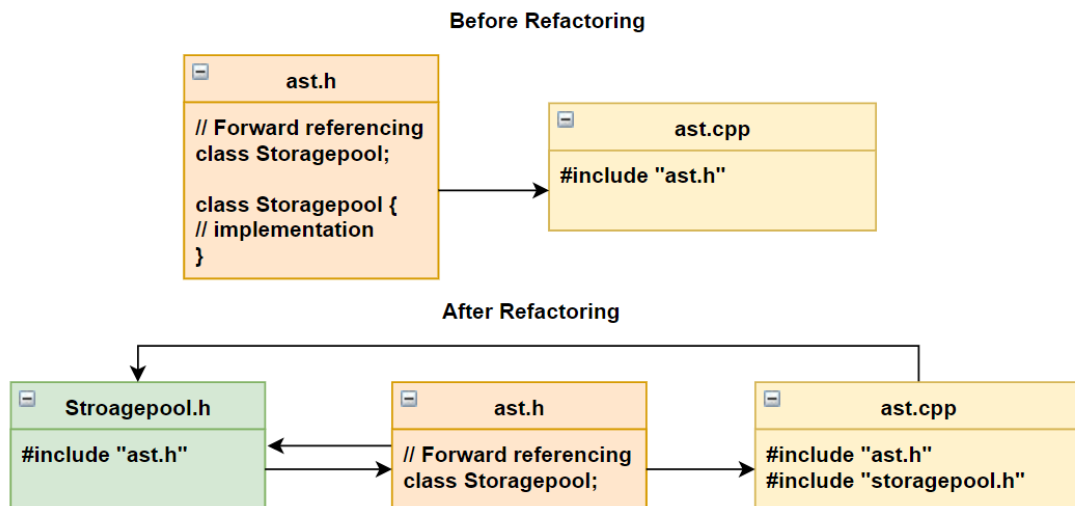


Figure 39: PINOT tangles due to cyclic dependencies.

6.4.2 Discontinued support for Jikes

Jikes is the core modules that facilitates PINOT to compile the java input files into PINOT understandable objects. But the support for Jikes has been discontinued as of 2010 and the last stable release of the software was released 16 years ago (current: 13-Jan-2021) [7]. It last supported the java files that were written to compile on the Java 5.0 platform. This indicates that PINOT is limited by Jikes and cannot understand the java source files that are compatible with the newer versions of Java.

One of the original creators has opened a new Github repository for Jikes[10]. The last commits to this repository are from April 2017 and indicate that this Jikes version has been updated more recently than the official version. This new version of Jikes may provide support for newer versions of Java. However additional research into this is required. Additional research is also required to review whether it is possible to use this new version of Jikes as a basis for PINOT as well.

6.4.3 Custom memory allocation

PINOT implements its variant of alloc functions that is responsible for allocation memory to its objects on the stack. This would have been feasible in older machines that was not equipped with stack canaries. Stack canaries are stack protectors that detect and prevent stack buffer overflow attacks from malicious code [11]. The custom allocation is detected as an potential stack overflow attack that terminates the PINOT's execution with segmentation faults.

6.4.4 Deep inheritance tree

PINOT exhibits deep inheritance within its codebase. Figure 40 depicts a rough outline of this issue. Every class that begin with the pattern Ast is a subclass of the super-most class Ast. Refactoring this structure is non-trivial and can induce unwanted tangles in the system.

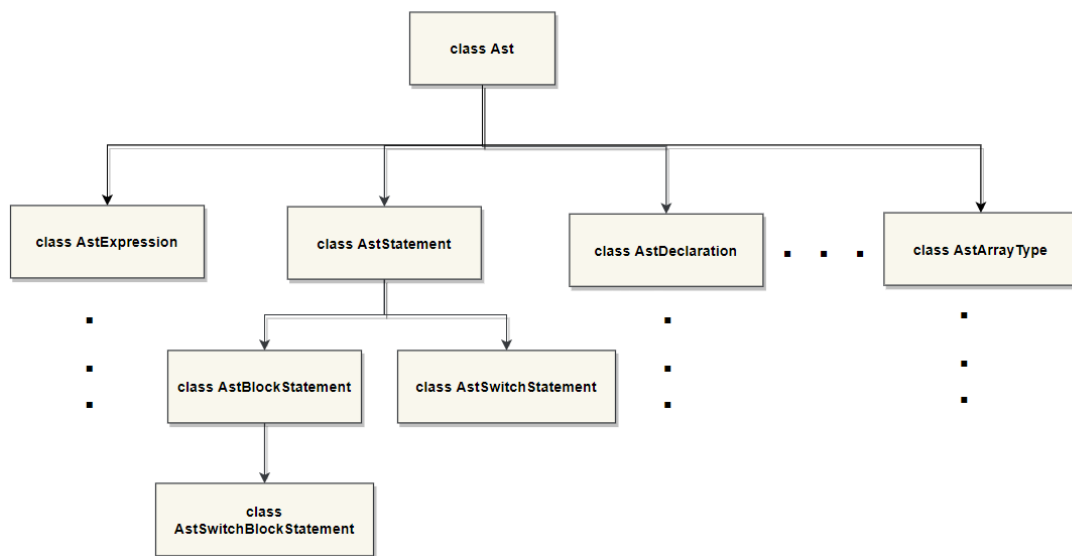


Figure 40: PINOT deep inheritance issue.

6.4.5 Incompatible compiler(s)

The reference reported provided by PINOT Java AWT 1.3, JHotDraw 6.0b1, Java Swing 1.4, java.io 1.4.2, java.net 1.4.2, javac 1.4.2, Apache Ant 1.6.2, ArgoUML 0.18.1 fail to run and result in segmentation faults. This could be due to incompatible compiler(s) or some missing header file. PINOT determines the environment in which it is currently executing and only utilizes the header files that are present in the system. There were no system requirements defined for PINOT or there was no information provided that relates to the environment that was used to generate these reference reports. Replicating the same result was not possible, so as an alternative we execute PINOT over a different reference codebase that contains all the GoF patterns as the reference report.

6.4.6 PINOT vs Modern Java code

After realizing the discontinuation of support of Jikes, we conducted a test of PINOT over the latest java syntax's. For this we picked a github repository that contained GoF patterns written in Java spring boot ¹⁰. Figure 41 shows that PINOT was unable to detect any modern patterns that

¹⁰<https://github.com/indrekru/design-patterns-spring-boot>

were depicted in the codebase. To support the modern java syntaxes, PINOT must be rebooted to be able to compile these newer syntaxes and appropriate java grammar rules has to be setup to convert the compiled source to PINOT understandable objects. For this an alternative to the Jikes compiler with active support has to be found.

```
Pattern Instance Statistics:

Creational Patterns
=====
Abstract Factory      0
Factory Method        0
Singleton             0
=====

Structural Patterns
=====
Adapter              0
Bridge               0
Composite            0
Decorator            0
Facade              0
Flyweight            0
Proxy               0
=====

Behavioral Patterns
=====
Chain of Responsibility 0
Mediator              0
Observer             0
State                0
Strategy             0
Template Method       0
Visitor              0
=====

Number of classes processed: 1
Number of files processed: 15
Size of DelegationTable: 0
Size of concrete class nodes: 0
Size of undirected invocation edges: 0

nMediatorFacadeDual/nMediator = 0/0
nImmutable/nFlyweight = 0/0
nFlyweightGoFVersion = 0
```

Figure 41: PINOT on modern Java syntax.

7 Future Research

Our refactoring goal was limited to a specific set of classes and files, the effects of the fat file refactoring appears small as PINOT is comprised of small but many class files. Refactoring all these classes will further improve the quality of PINOT.

PINOT is primarily limited by the discontinued support for its Jikes compiler, a core component in its making. It is essential to find and integrate a suitable alternative to reboot PINOT to recognize the newer Java syntax's and patterns and appropriate future-proofing strategies must be adopted to evolve PINOT alongside the evolution of Java itself.

JikesRVM¹¹ is an **active** java compiler written in C or C++. The compiler currently supports Java 9 which is a big improvement from the IBM Jikes which is at java 5.0, this is also an indication that most modern java syntaxes can be detected using the JikesRVM compiler. The last update to this repository was made 19 days ago as of 17-January-2021 indicating that the developers are actively updating the repository. Incorporating the JikesRVM in place of the legacy Jikes can be a potential optimistic extension path for PINOT. The only risk involved in this approach is the development of the PINOT will be limited by the development of JikesRVM. Nevertheless, this approach should conceptually lead to a substantially lower amount of rework required to modernize PINOT.

The parsing functionality of PINOT is controlled by defining Java grammar. It is essential that if an alternate compiler is found for PINOT there should be provisions provided to modify the nature of the compiler similar to the Jikes compiler.

PINOT currently only supports Java source code as its input, which in our opinion is wasted potential. Any programming language that can be translated into an Abstract Syntax Tree should be a potential opportunity for PINOT to expand its domain.

An alternate approach is to redesign the PINOT using a Java or similar modern programming platform. This ensures that the programming platform of PINOT would be native and can leverage the power of modern Java compilers ensuring concrete future-proofing. Another approach to this is the use of **Named pipes** that can be used to transfer data from Java and C++. The compilation part can conceptually be abstracted to be performed natively, the data can then be piped to a newly developed module and/or adapter in PINOT that can translate this to PINOT understandable objects.

¹¹<https://github.com/JikesRVM/JikesRVM>

8 Conclusion

PINOT can be used to generate wonderful insights into the pattern used in existing Java applications. The combination of techniques and algorithms used for this purpose is powerful and quite unique. It is even more unique since PINOT was designed over 15 years ago and still offers added value in the process of application comprehension. The age of PINOT does however provide the first challenge as compiling and using PINOT requires the right and often old versions of compilers and other related packages.

Another challenge of PINOT is indicated based on an elaborate analysis of the source code. PINOT's codebase comprises over 100 thousand lines of code, which is not necessarily a problem. However, it does become a problem when a lot of these lines and logic are put into few files as is the case in the architecture of PINOT or better the base of PINOT, Jikes. Thereby the architecture of PINOT could be described as fat, which made it fairly difficult to understand the responsibility and functionality of different files and classes in the system. Combine this with a major amount of forward references and circular dependencies and it is almost impossible to separate files, classes, or types without introducing tangles.

A refactor plan for PINOT has been created, applied, and evaluated to improve the architectural structure of PINOT. The newly refactored architecture of PINOT reduced the fatness of the architecture by approximately 50 percent. While this is a substantial improvement the number of circular dependencies increased drastically as well. While an increase in tangles was expected, an increase of the observed magnitude was not anticipated beforehand.

All in all, this report presents an analysis and refactored architecture for PINOT. While the refactored architecture still contains flaws we believe it is a first step in improving the structure of PINOT.

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] N. Shi and R. A. Olsson. "Reverse Engineering of Design Patterns from Java Source Code". In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 2006, pp. 123–134. DOI: [10.1109/ASE.2006.57](https://doi.org/10.1109/ASE.2006.57).
- [3] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1995.
- [4] 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.
- [5] 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.
- [6] Understand by Scitools. *Understand Metrics list documentation*. <https://support.scitools.com/t/what-metrics-does-understand-have/66>. [Online; accessed 08-December-2020]. 2020.
- [7] Wikipedia contributors. *Jikes* — *Wikipedia, The Free Encyclopedia*. <https://en.wikipedia.org/wiki/Jikes>. [Online; accessed 27-December-2020]. 2020.
- [8] Wikipedia contributors. *Control flow* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Control_flow. [Online; accessed 17-January-2021]. 2020.
- [9] Wikipedia contributors. *Circular dependency* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Circular_dependency. [Online; accessed 13-January-2021]. 2020.
- [10] Dave Shields. *Jikes*. <https://github.com/daveshields/jikes>. [Online; accessed 13-January-2021]. Apr. 2017.
- [11] Wikipedia contributors. *Stack Buffer overflow* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Stack_buffer_overflow. [Online; accessed 13-January-2021]. 2021.

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.

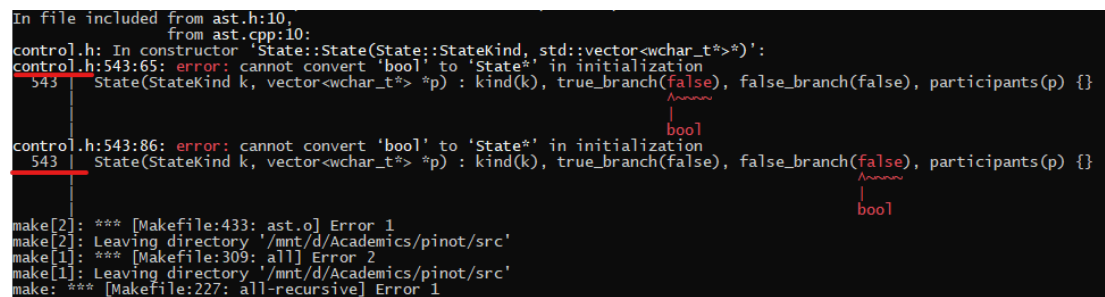
A.1 Building PINOT

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



```
In file included from ast.h:10,
                 from ast.cpp:10:
control.h: In constructor 'State::State(State::StateKind, std::vector<wchar_t*>)':
control.h:543:65: error: cannot convert 'bool' to 'State*' in initialization
543 | State(StateKind k, vector<wchar_t*> *p) : kind(k), true_branch(false), false_branch(false), participants(p) {}
    |                                                         ^~~~~~
    |                                                         |
    |                                                         bool
control.h:543:86: error: cannot convert 'bool' to 'State*' in initialization
543 | State(StateKind k, vector<wchar_t*> *p) : kind(k), true_branch(false), false_branch(false), participants(p) {}
    |                                                         ^~~~~~
    |                                                         |
    |                                                         bool
make[2]: *** [Makefile:433: ast.o] Error 1
make[2]: Leaving directory '/mnt/d/Academics/pinot/src'
make[1]: *** [Makefile:309: all] Error 2
make[1]: Leaving directory '/mnt/d/Academics/pinot/src'
make: *** [Makefile:227: all-recursive] Error 1
```

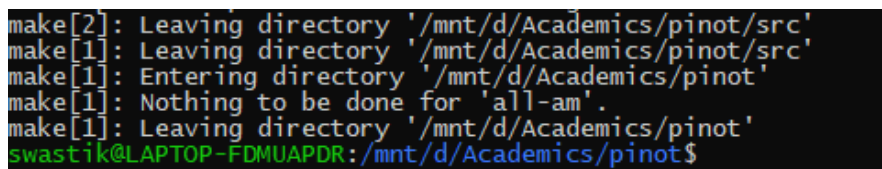
Figure 42: cannot convert 'bool' to 'State*' in initialization Error.

From figure 42, 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.

```
1 State(StateKind k, vector<wchar_t*> *p) : kind(k), true_branch(false),
2                                           false_branch(false),
3                                           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 43 after applying this fix.

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



```
make[2]: Leaving directory '/mnt/d/Academics/pinot/src'
make[1]: Leaving directory '/mnt/d/Academics/pinot/src'
make[1]: Entering directory '/mnt/d/Academics/pinot'
make[1]: Nothing to be done for 'all-am'.
make[1]: Leaving directory '/mnt/d/Academics/pinot'
swastik@LAPTOP-FDMUAPDR: /mnt/d/Academics/pinot$
```

Figure 43: Build success after the fix.

A.2 Building .cpa files for Structure101

```
in file included from /mnt/d/Academics/pinot/src/control.h:15:  
/mnt/d/Academics/pinot/src/platform.h:41:10: error: 'config.h' file not found with <angled> include; use "quotes" instead  
#include <config.h>  
~~~~~  
"config.h"  
1 error generated.
```

Figure 44: Config.h build error, generating .cpa file for Structure101.

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.

```
--CPA generation from ASTs completed--  
2020-11-30 19:22:43 INFO - 39 files processed
```

Figure 45: cpa file generated message, after fix to platform.h.

The message displayed in Figure 45 should now be printed on your console once the script has finished execution.

B Figures

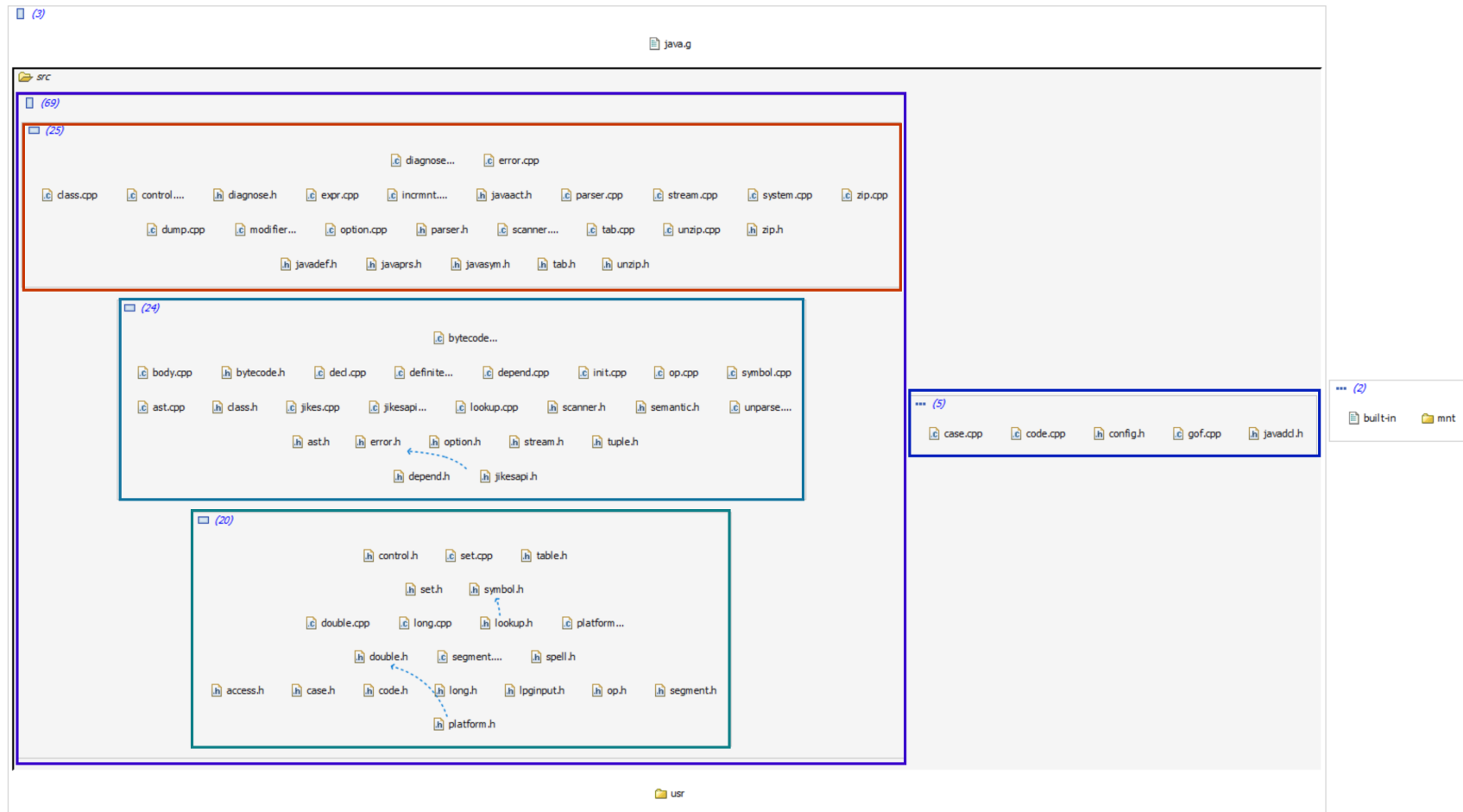


Figure 46: PINOT source code cohesive clusters as reversed engineered by Structure101.

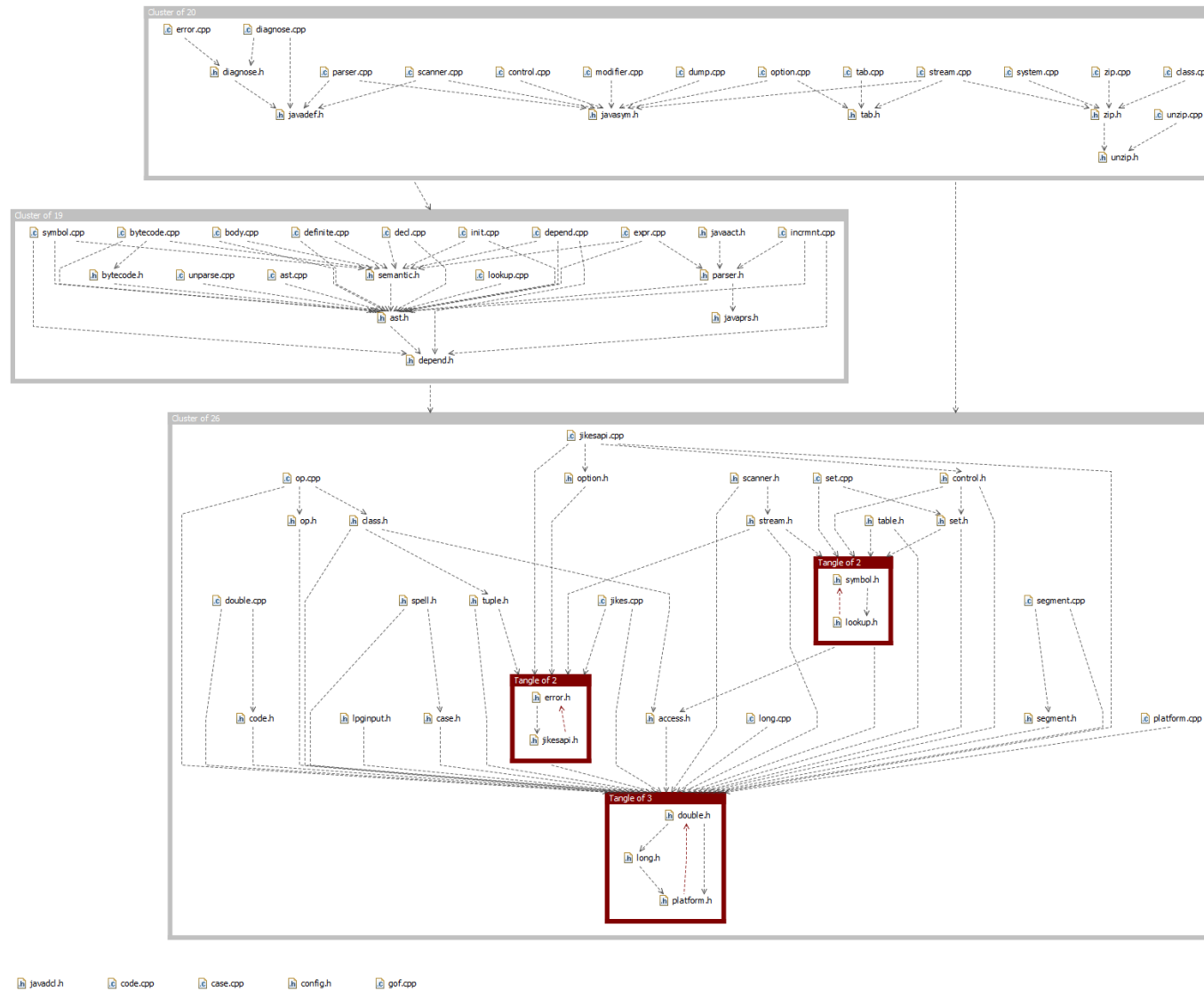


Figure 47: Dependencies between files in PINOT

B.1 Object-Oriented Metrics Analysis

B.1.1 PINOT OOM Analysis

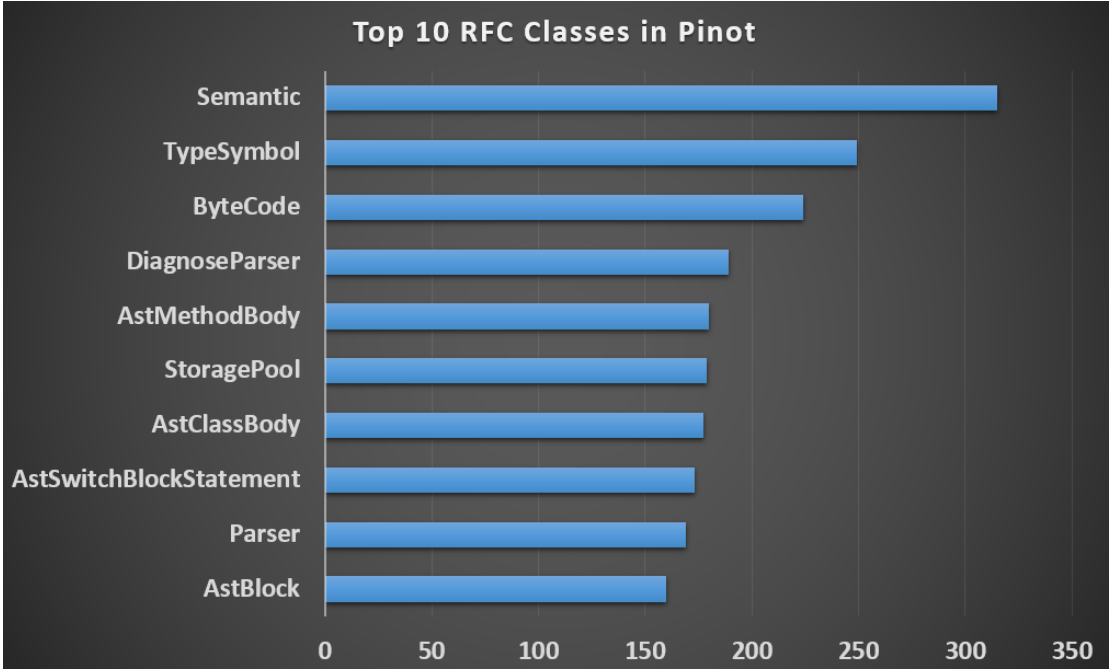


Figure 48: Top 10 RFC classes in PINOT.

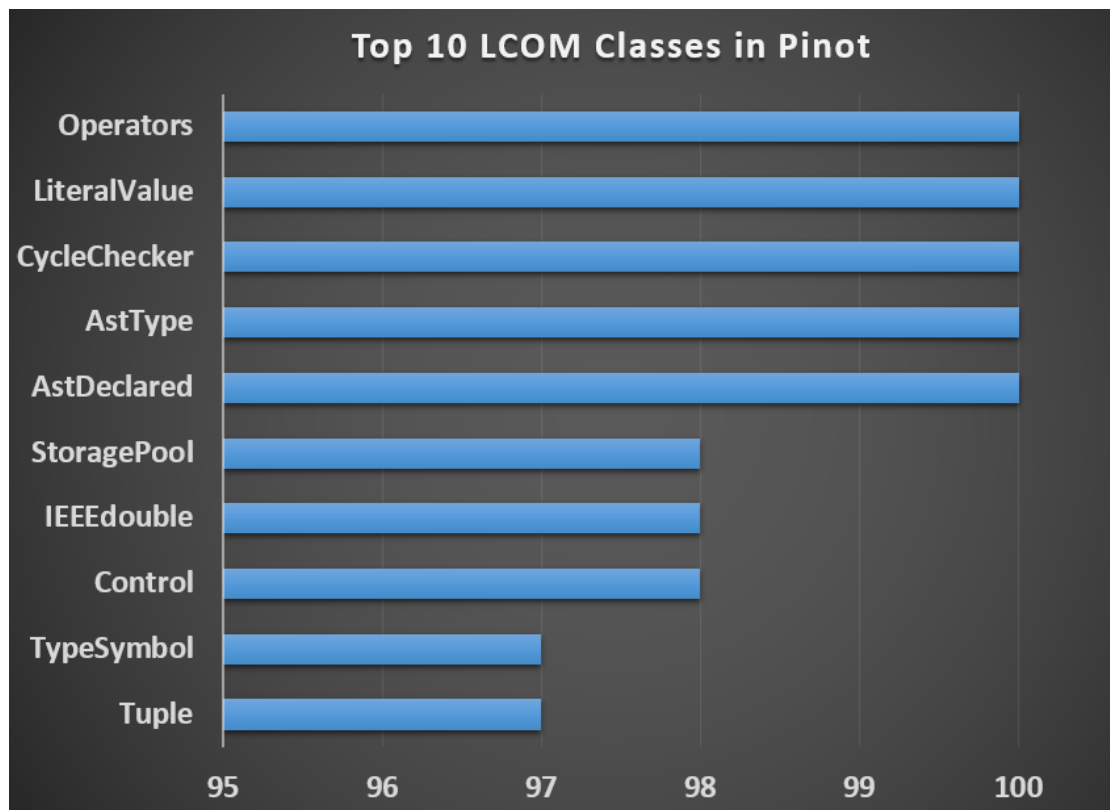


Figure 49: Top 10 LCOM classes in PINOT.

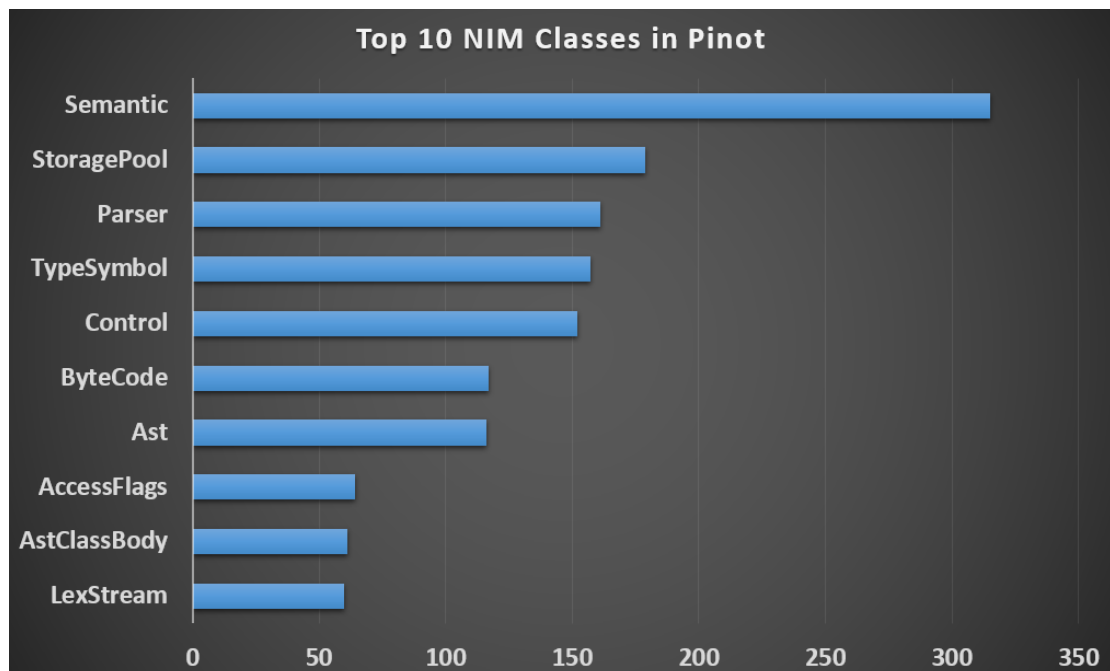


Figure 50: Top 10 NIM classes in PINOT.

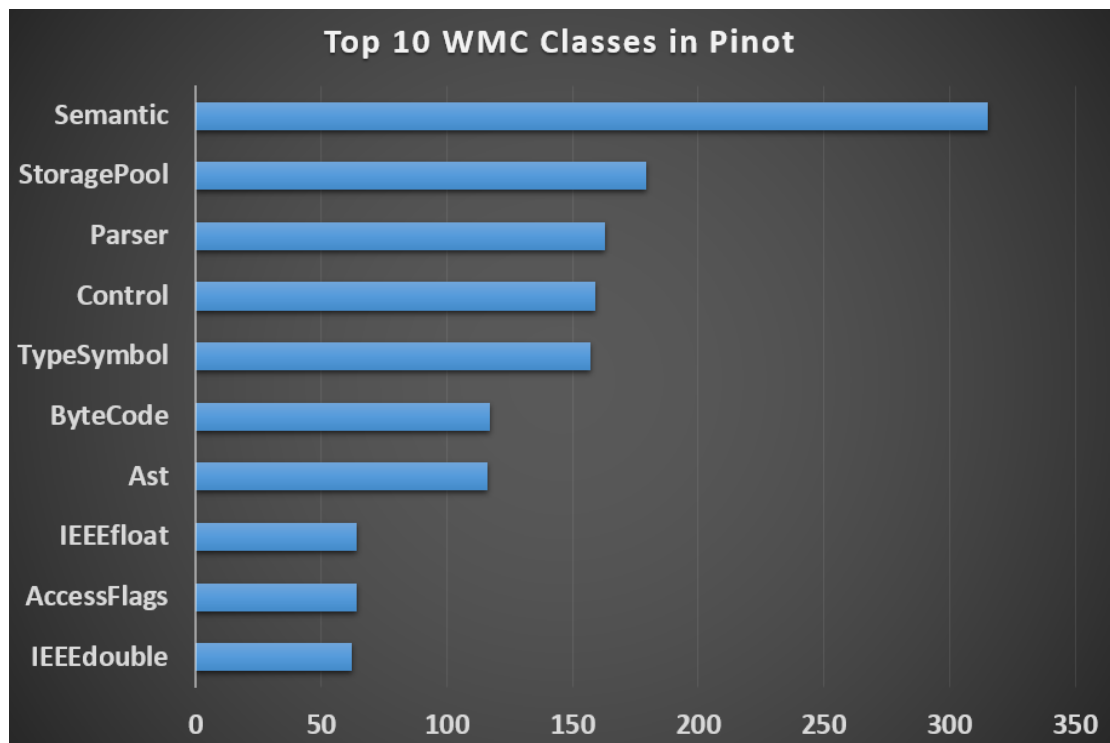


Figure 51: Top 10 WMC classes in PINOT.

B.1.2 Ast OOM Analysis

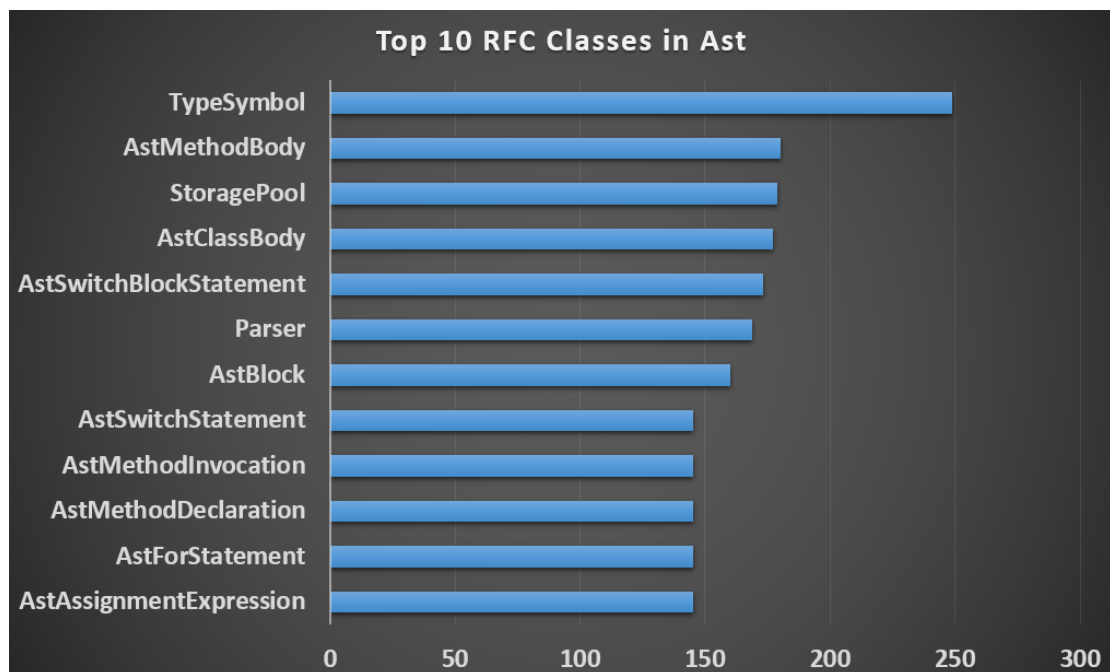


Figure 52: Top 10 RFC classes in Ast package.

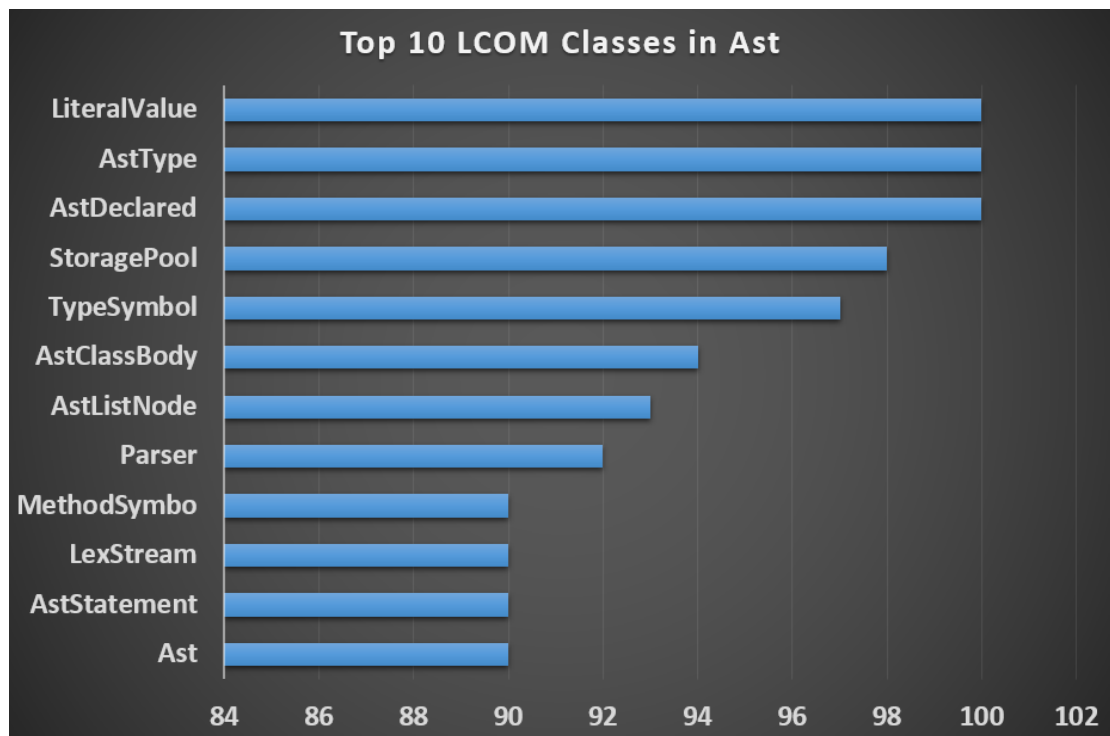


Figure 53: Top 10 LCOM classes in Ast package.

B.1.3 Symbol OOM Analysis

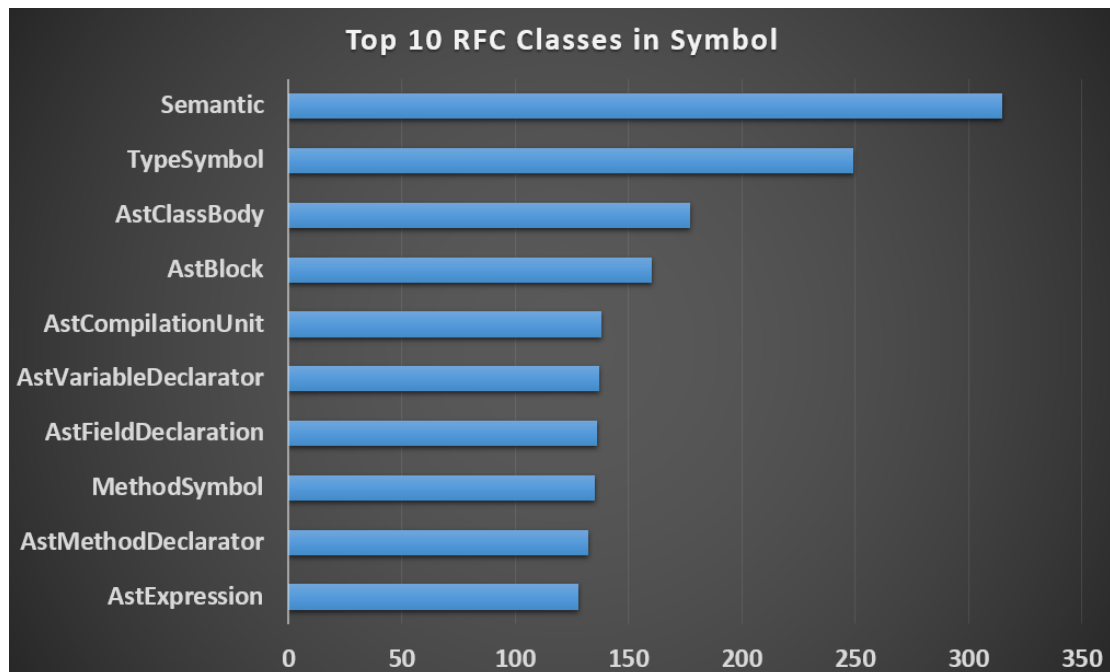


Figure 54: Top 10 RFC classes in Symbol package.

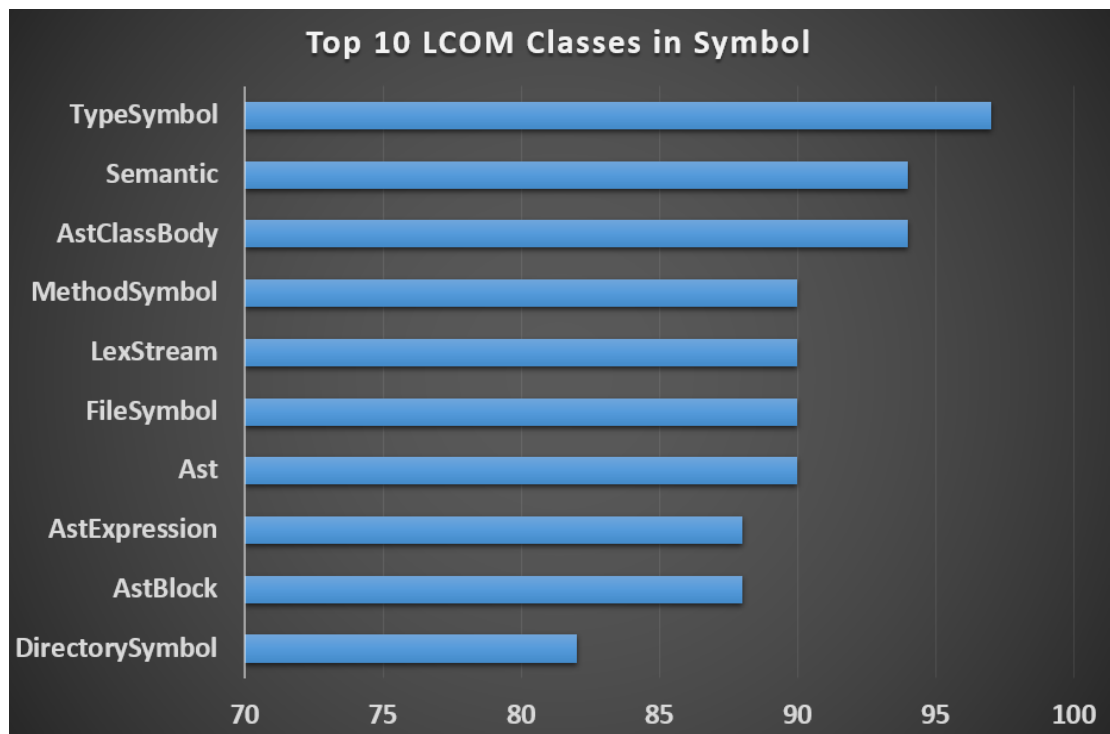


Figure 55: Top 10 LCOM classes in Symbol package.

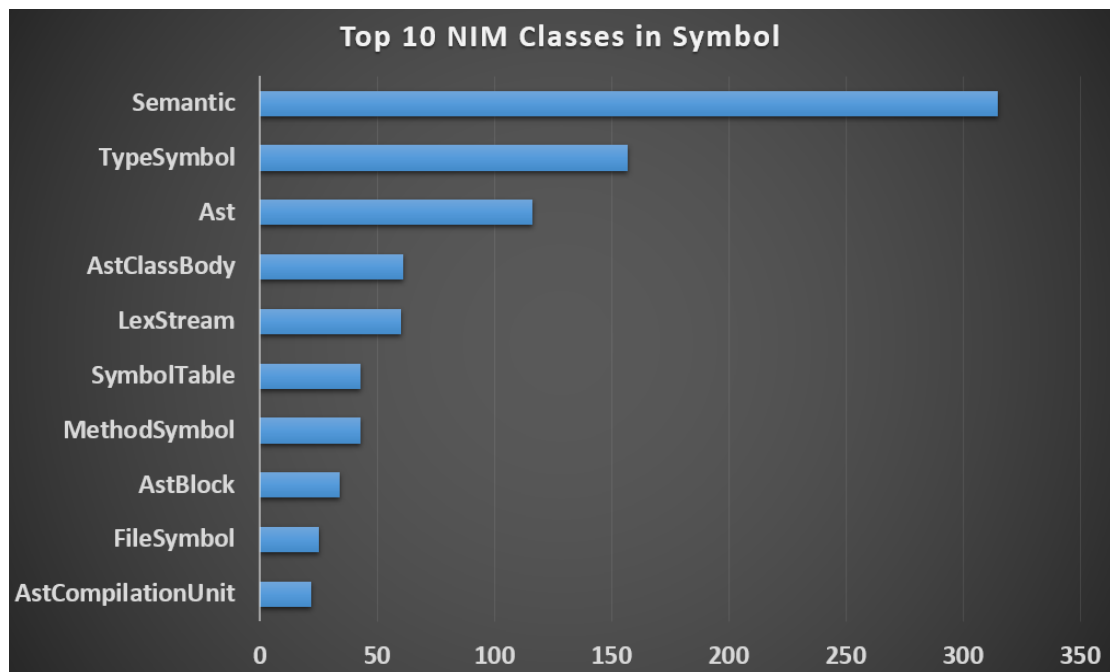


Figure 56: Top 10 NIM classes in Symbol package.

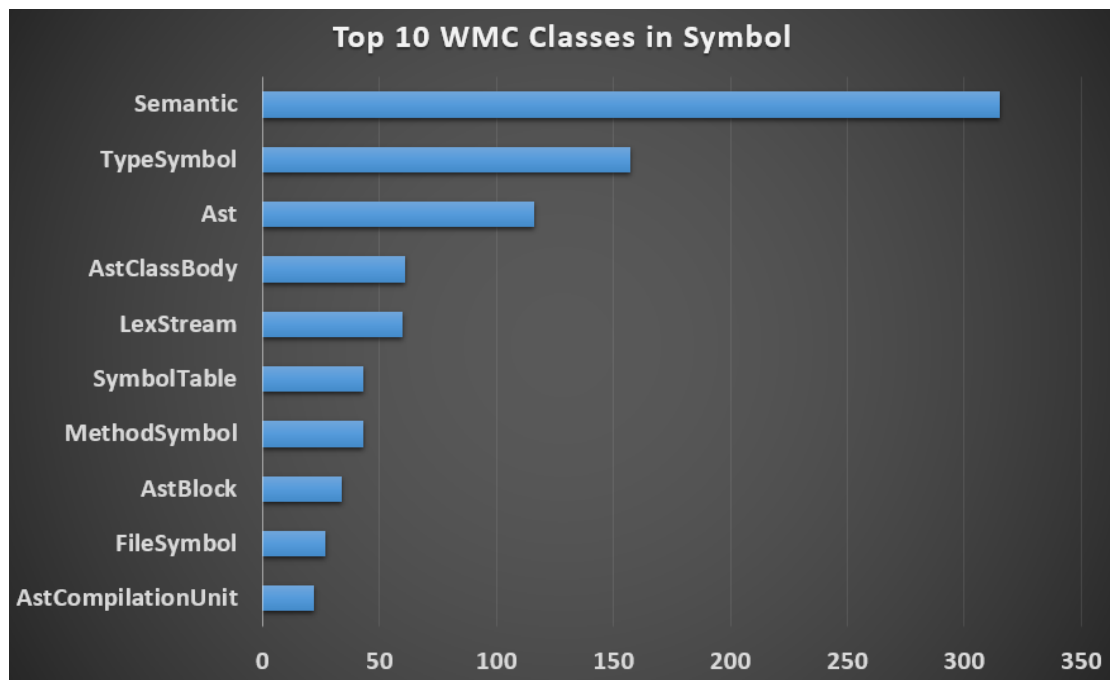


Figure 57: Top 10 WMC classes in Symbol package.

B.1.4 Control OOM Analysis

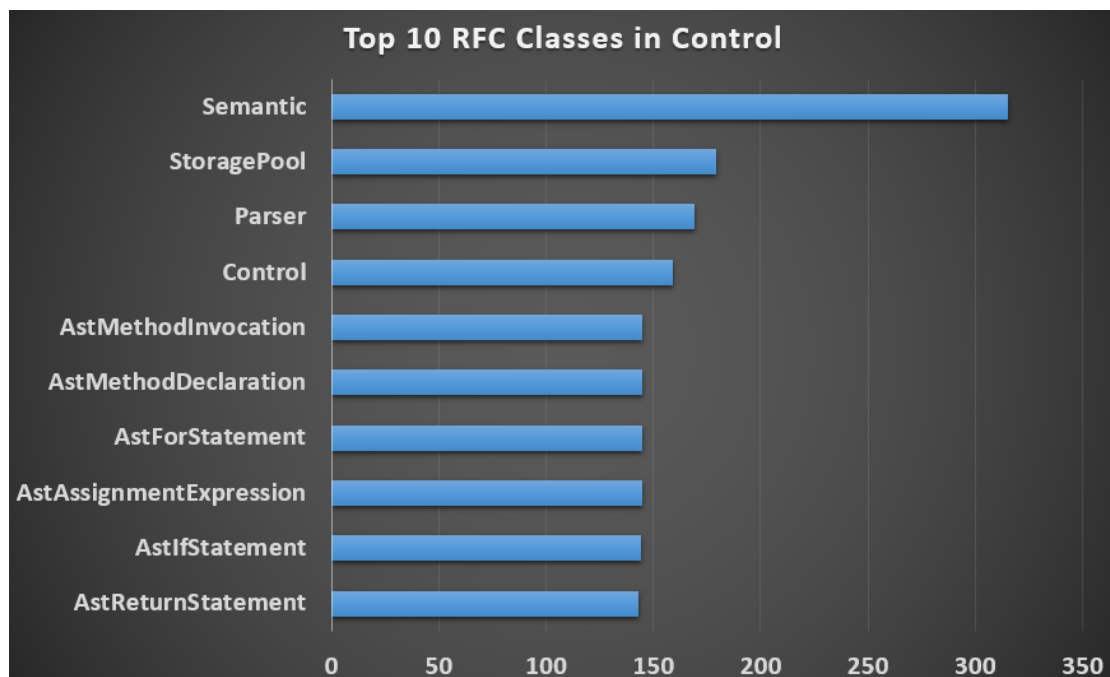


Figure 58: Top 10 RFC classes in Control package.



Figure 59: Top 10 LCOM classes in Control package.

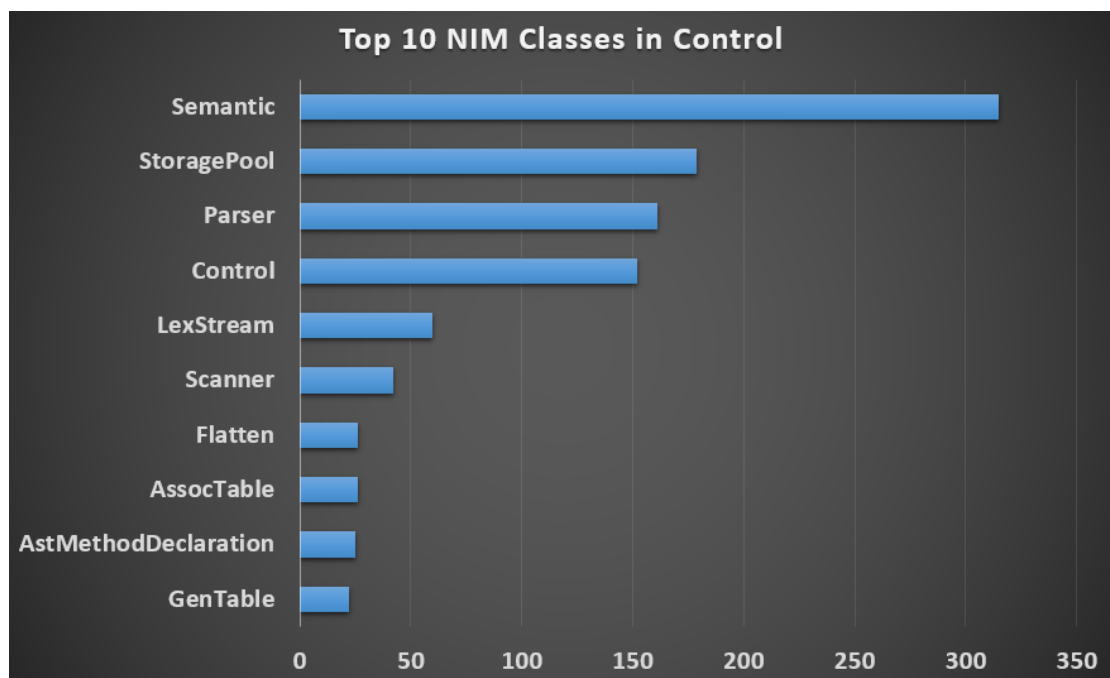


Figure 60: Top 10 NIM classes in Control package.

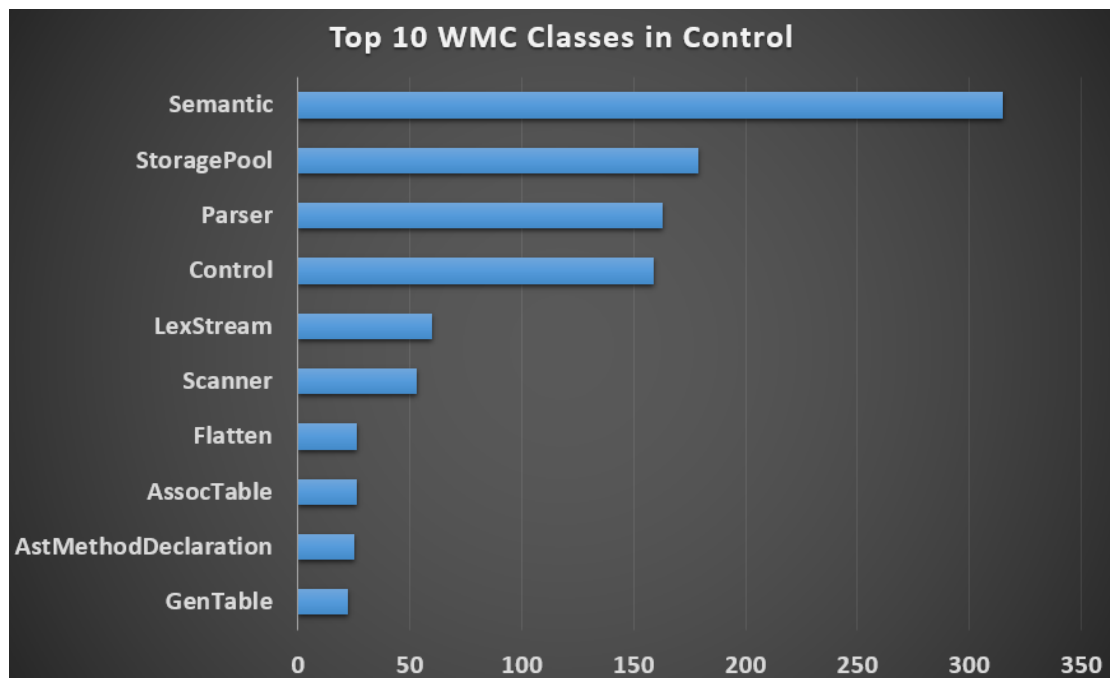


Figure 61: Top 10 WMC classes in Control package.

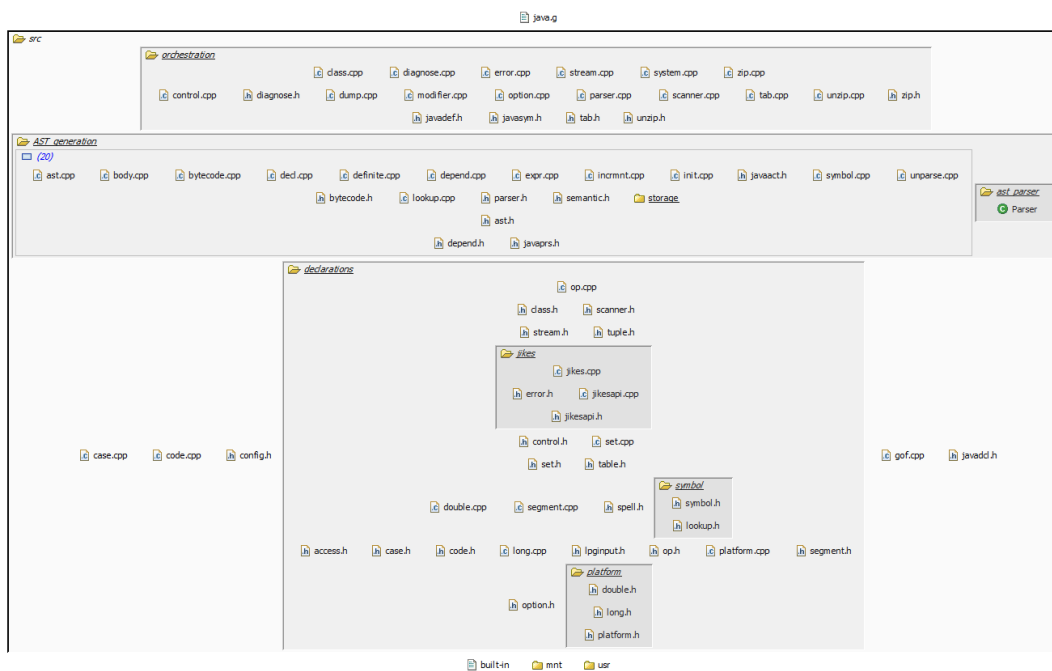


Figure 62: Refactoring plan to reduce fat folders in PINOT.

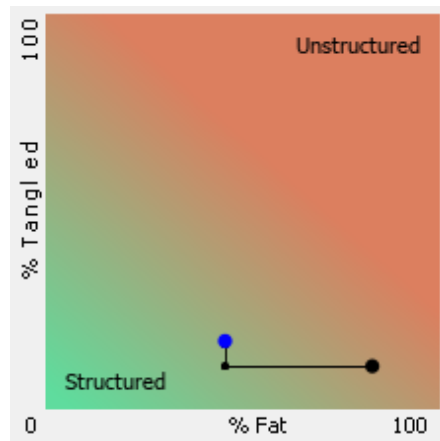


Figure 63: Structure complexity of PINOT for fat folder refactoring plan.

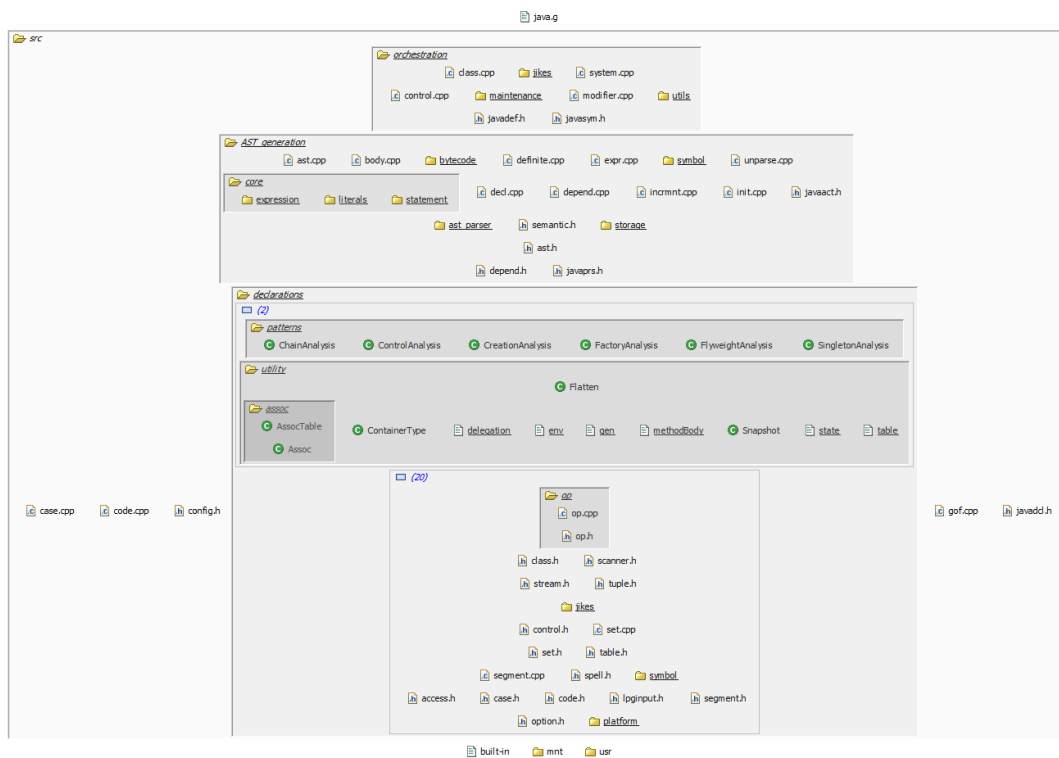


Figure 64: Refactoring plan to reduce fat files in PINOT.

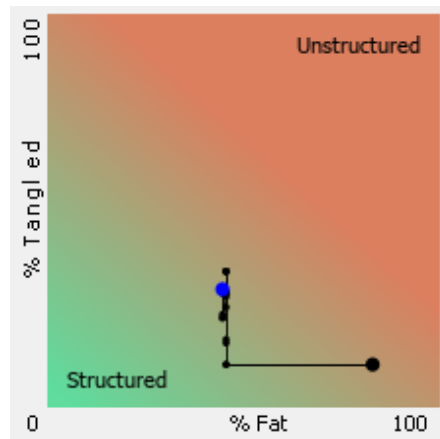


Figure 65: Structure complexity of PINOT for fat files refactoring plan.

C Relative Comparison

This appendix describes the relative comparison that has been conducted between the Structure101 complexity metrics of the original architecture and the refactored architecture. This comparison has been conducted because the fat folder metric had a strong influence(74%) in the original architecture and none influence(0%) in the refactored architecture. This large drop in influence made it difficult to compare the other metrics. In table 1, the influence of each metric on the complexity is indicated. The column Relative Original Architecture indicates the influence of all metrics without the influence of the fat folder metric in the original architecture.

Metric	Original Architecture	Relative Original Architecture	Refactored Architecture
Fat Folder	74	0	0
Fat File	18	69,23076923	39
Fat Type	0	0	5
Fat Function	6	23,07692308	20
Tangles Folder	2	7,692307692	36

Table 1: The table shows the influence of each metric on the complexity in the different situations. Where the Relative Original Architecture situation indicates the influence of all metrics after excluding the influence of Fat Folder in the original situation.

D 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 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 5.
		9/12/2020	Added section 5.2.2.
	SN	8/12/2020	Working on Section 3.6 and Appendix B.1. Image enhancement for Figure 4.
		9/12/2020	Finished Section 3.6 and Appendix B.1. Section rework 3.3.
4.0	SO	9/12 to 16/12/2020	Worked on code refactoring.
	SN	11/12 to 15/12/2020 16/12/2020	Worked on code refactoring. Added Section 4 and feedback rework.
5.0	SO	6/01/2021	Worked on Section 5 and Appendix C. Worked on Section 4
	SN	6/01/2021 16/12/2020	Worked on Section 4. Worked on Section 5 and Appendix C.
6.0	SO	13/01/2021	Worked on Sections 4, 5, 6, 7, and 8.
	SN	13/01/2021	Worked on Sections 4, 5, 6, 7, and 8.
7.0	SO	14/01/2021 to 17/01/2021	Report rework and incorporated feedback.
	SN	14/01/2021 to 17/01/2021	Report rework and incorporated feedback.