Dependency Graph Creator Engine

Software Maintenance and Evolution

https://github.com/richardswesterhof/pyne

Authors:
Job Heersink (s3364321)
Richard Westerhof (s3479692)

Supervisors:
Mohamed Soliman
Filipe Capela

University of Groningen
The Netherlands
January 16, 2021

Contents

Chapter 1

Introduction

Software maintenance has always been of great concern to software engineers. An application is almost never perfect on deployment and needs regular bug-fixes and enhancements to the system. To help with the software maintenance of an application, several different techniques have been developed.

One of these techniques is a dependency graph. A dependency graph is a directed graph that represents dependencies between objects of some application domain, where the nodes represent packages or classes of a software system and the edges the relation between them.

Many different programs have been created to generate such a dependency graph from source files or executables. Although a lot of options are available, they all still seem to struggle to get consistent, or even complete, results [1]. This would point to the fact that there is still a lot of room for improvement in the field of dependency graph creation.

In this document we will analyse and try to improve an incomplete existing dependency graph creator called Pyne [2]. The main goal here is to improve Pyne is such a way that it is able to see more dependencies, since it is currently missing a few as we will show later on. We will try to see if our changes improved this program by comparing it to another widely used dependency graph creator engine called Structure101 [3] and running both these engines on Apache Tajo [4]. To facilitate this comparison, we will create a dependency checker program that will compare the output of both dependency graph creator engines.

The document is structured as follows: In chapter 2 we will show how Pyne, Structure101 and the dependency checker should be installed and used. In chapter 3 we will discuss the design of Pyne itself. In chapter 4 we will analyse Apache Tajo [4] and compare the output of Pyne and Structure101 on that program. In chapter 5 we will discuss our dependency checker program that we created to evaluate the outputs of the two dependency checkers. In chapter 6 we will discuss the problems of Pyne, how we tried to fix them and analyse how well these changes improved the program and finally in chapter 7 and chapter 8 we will discuss and conclude the project, respectively.

Chapter 2

User Guide

Contents

In this chapter...

In this chapter we will describe the process to get all tools up and running step by step, with example commands provided as well. This includes Pyne and Structure101, as well as our own dependency checker.

2.1 Running Pyne

Installing

To install Pyne, Java JDK 11+ and Maven is required. The installation itself is rather straightforward: Clone or download our fork of the Pyne repository:

```
git clone https://github.com/richardswesterhof/pyne.git
```

Install Pyne's dependencies and build an executable jar by running the following command inside Pyne's root directory:

```
mvn install
```

Executing

After the application has been built, this application can be used via CLI. This can be done by running the following command from the root directory:

```
java -jar <path-to-jar>/pyne-cli-1.0-SNAPSHOT-jar-with-dependencies.jar
  ↪ <github-url>
```

Where the `<path-to-jar>` is the path to the built jar file (by default this is in `<pyne_root>/pyne-cli/target/`) and `<github-url>` is the url of the project you want to create a dependency graph for. For example, this would be the command to create a yearly dependency graph for the commits of Apache Tajo from 2015 to November 2020:

```
java -jar pyne-cli/target/pyne-cli-1.0-SNAPSHOT-jar-with-dependencies.jar
  ↪ https://github.com/apache/tajo -s "2015-01-01" -e "2020-11-21" -p
  ↪ "YEAR"
```

2.2 Running Structure101

To generate the output from Structure101, follow the steps below:

1. Make sure Structure101 is installed.
2. Create a new project, select "Maven" as the option for discovering bytecode, select the pom file of the project you want to analyse, and make sure to check the "Parse Tests" and "Parse Profiles" boxes to include as many sources as possible.
3. Now you should already be able to select the way you want everything to be organized, i.e. whether you want packages, leaf packages or classes. If you only want to analyse one of the two, you can select this now (note that for analysing on a package level, make sure to select "Leaf packages" instead of just "package"). However, if you want to analyse both, it is easier to select "Packages" here.
4. Select the "Detail" granularity, and check "Included injected dependencies" to again cover as many dependencies as possible.
5. Select "Show externals" so that external packages are shown as well as internal packages, and check "Parse archives contained in classpath archives".
6. Leave the list of exclusions empty, and select the project you want to analyse's folder as the sources.

Now the project is set up, and the results can be exported to CSV. If there are any issues during the setup of the project, there is a Structure101 project for Tajo available on our GitHub repository[1], and Structure101's outputs on Tajo using the settings in the instructions above are also available.

7. To generate the expected matrix from Structure101, right-click anywhere outside the packages in the main graph view, hover over the "Flatten" option, and select "To leaf packages" (for comparing on package level) or "To classes" (for comparing on class level) from the sub-menu (skip this step if you already did this during the project setup).
8. Go to the "View" tab and select the composition view. You should now see the matrix that shows the amount of dependencies between each class/package.
9. A CSV file must be exported by clicking the export icon in the top right toolbar in the matrix panel. Make sure that you select "file" for the "Export to" option, then select "Matrix as CSV" in the "Export as" dropdown. Finally, select a file location to save the file to in the "Target file" field, and click "Ok" to export.
10. (optional) If you want to do an analysis on both class and package level, go back to the main view and use control+z to undo the flattening, then return to step 7.

[1] `https://github.com/richardswesterhof/pyne/blob/master/structure101/tajo_structure101.java.hsp`

Now you have the output file necessary from Structure101. This is the file you should use when using the dependency checker.

2.3 Running the dependency checker

To use the dependency checker, Java JDK 11+ and Maven is required. The installation itself is rather straightforward: Clone or download our fork of the Pyne repository. The dependency checker will be in there too.

```
git clone https://github.com/richardswesterhof/pyne.git
```

Install the dependency checker's dependencies and build an executable jar by running the following command inside the `dependency_checker/` folder:

```
mvn install
```

This will place a ready-to-execute jar file in the `target/` subfolder. Alternatively, a pre-built jar file is already located in the `dependency_checker/` folder.

Executing
After the application has been built, one can use this application via CLI. This can be done by running the following command from the directory where the jar file is located:

```
java -jar dependency_checker.jar -s [PATH/TO/STRUCTURE101_FILE.csv] -p
  ↪ [PATH/TO/PYNE_FILE.graphml] -d [CLASS|PACKAGE] [OPTIONS]
```

With `-d` you can specify whether you want to compare the dependencies between packages or classes, with `-p` you must specify the path to the graphml output-file of Pyne and with `-s` you must specify the path to the Structure101 CSV file. This file must either contain the relation of the classes or the packages depending on the specified option for `-d`. If the CSV file contains class relations, while `-d` is set to package, the dependency checker will not work. The other available options include:
The `-hr` option will create a human readable version of the output (with tabs and newlines, and more explicit definitions, so cross referencing IDs is not necessary). This is good for manually investigating the results.
The `-i` option will only indent the file (and place newlines), the `-hr` options implies this as well.
The `-c` option will generate a compact output, which aims to reduce duplicate data as much as possible. This is good for programmatically investigating the results.
Note that the options `-hr` and `-c` cannot be used together.

Given below is an example of how the dependency checker could be run:

```
java -jar dependency_checker.jar -s
  ↪ ../graph-files/tajo_dependencies_by_structure101.csv -p
  ↪ ../graph-files/tajo_dependencies_by_pyne.graphml -hr -d PACKAGE
```

Chapter 3

Design

Contents

In this chapter...

In this chapter we will go into more detail about the design of Pyne itself. The project has been analysed using the existing documentation [2], the source code and the structural and dependency graphs created by Structure101 (figure 3.2).

3.1 Requirements

According to the documentation [2] of the Pyne project, the project has the following requirements:

1. **The program should be able to create a graph from Java source files.**
2. The output graph should be the same as that of Arcan[1], a Java software analysis tool.
3. The program should parse Java source files in such a way so it can find the different Java classes and packages.
4. The program should be able to retrieve the source files using Git.
5. The program should provide a command line interface.

When inspecting the functionality of the program, one can see that all these requirements have been met. When inspecting the source code of the program and the documentation, one can even see how they implemented the requirements.

From the documentation it was made clear that most of the requirements were met using external dependencies. For example: Requirement 2 was met using Apache Tinkerpop[2] for graph creation, which is the same technology Arcan uses.
Requirement 3 was met using Spoon[3], an extensive Java source code parsing library. Lastly, Requirement 4 was met using Jgit[4], a java git library.

In addition to that, using Structure101, we found that Requirement 5 is met using Apache Commons Cli[5] to provide a cli interface. More information on this can be found in section 3.3.

3.2 Implementation

This application is divided into 3 different packages: `pyne-demo`, `pyne-cli` and `pyne-api`. The relations between these packages and their classes can be seen in figure 3.2. Figure 3.1 represents the XS (”excess”) diagram of Pyne [5]. The XS diagram reflects the size of a method, package, class or other code item. It will ensure that overly complex high level packages will have higher XS than a single method. This will reflect the likely negative impact on development. As you can see from the diagram, there are two components used to measure the code items: fat and tangled. Fat measures the amount of basic complexity and tangled measures cyclic dependency. Tangled is mostly applied to higher level components like design and package.

Looking at figure 3.1, we can see that the fat of the methods in Pyne is relatively small.

[1]https://essere.disco.unimib.it/wiki/arcan/
[2]https://tinkerpop.apache.org/
[3]http://spoon.gforge.inria.fr/
[4]https://www.eclipse.org/jgit/
[5]https://commons.apache.org/proper/commons-cli/

This means that the methods in Pyne are overall of a good size. However, looking at the fat of the class and packages, we see that they are on the larger side. To improve this, existing classes and packages could be split up into multiple classes/packages. Looking at the design, we see that it is not really fat, but extremely tangled. This means that there are a lot of cyclic dependencies. Most of the time cyclic dependencies can have a negative impact on future development and maintenance of the application, so they should be avoided as much as possible.

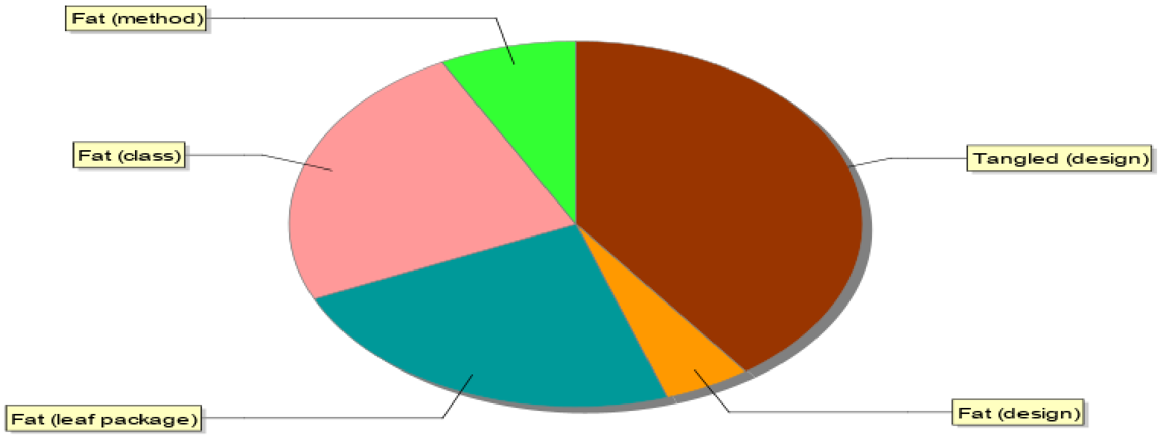

Figure 3.1: The XS diagram of the structure of Pyne, created by Structure101

Now that we have looked at the overall fat and tangles of Pyne, we will briefly explain the 3 main packages within Pyne. The `pyne-api` package will be explained in more detail, since this is the main component of the system.

3.2.1 Pyne-demo

This package acts as a testing ground for Pyne. Most of the data in this package has been hard-coded to work only with a specific repository that was probably present on the creators local machine. Since we do not know which repository they used for testing, this package will not be able to work and we will consider this package deprecated.

3.2.2 Pyne-cli

This package acts as a layer of communication between the user and the Pyne API via a CLI (Command Line Interface). Given a Git repository as a program argument, it will automatically start communicating with the Pyne API and create a dependency graph in graphml format for each commit. A number of options can be given to the CLI program. The first required argument is of course the repository itself, but one can also specify the starting and end date to parse from, the interval between each commit and an input and output directory.

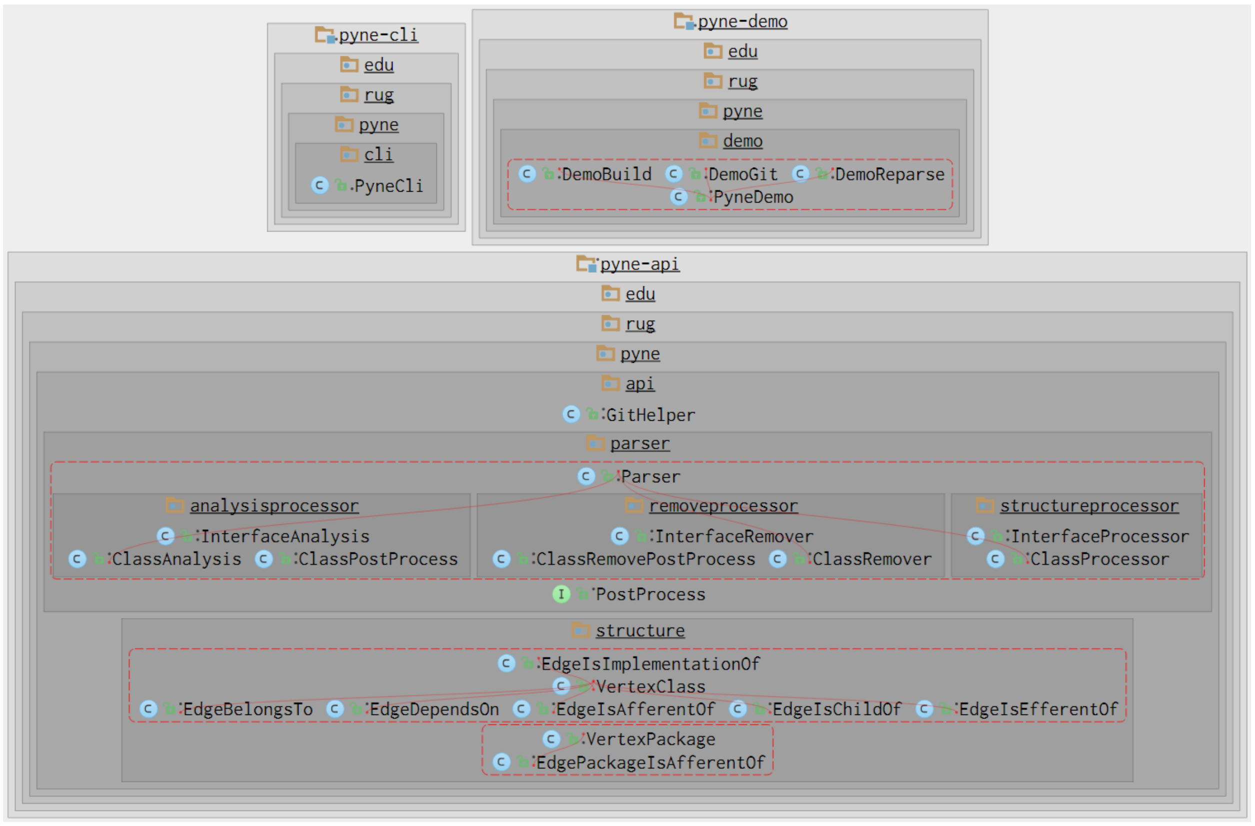

Figure 3.2: The full graph of the relations between the classes of Pyne, created by Structure101

3.2.3 Pyne-api

This package is the heart of the program and is the one that is actually responsible for the dependency graph creation. This package is again subdivided into 2 packages: `parser` and `structure`, and contains one class: `GitHelper`, as can be seen in figure 3.3.

The GitHelper is a part of the API responsible for Git related functionality like cloning a repository in a temporary location, parsing a commit or processing a commit. Most of the functionality of the GitHelper depends on the parser class to create the dependency graph.

The `structure` package does not contain any functionality, instead it provides abstract classes to build a dependency graph from. Special edges have been created for different kinds of relations: BelongsTo, DependsOn, AfferentOf, ChildOf, EfferentOf, ImplementationOf and PackageisAfferentOF. Special vertices have also been created to represent either a package or a class.

The `parser` package again consist of three sub-packages: `analysisProcessor`, `removeProcessor`, and `structureProcessor`. It also contains the `Parser` class and the `PostProcess` interface.

All communication with this package goes through the Parser class. This class is used to

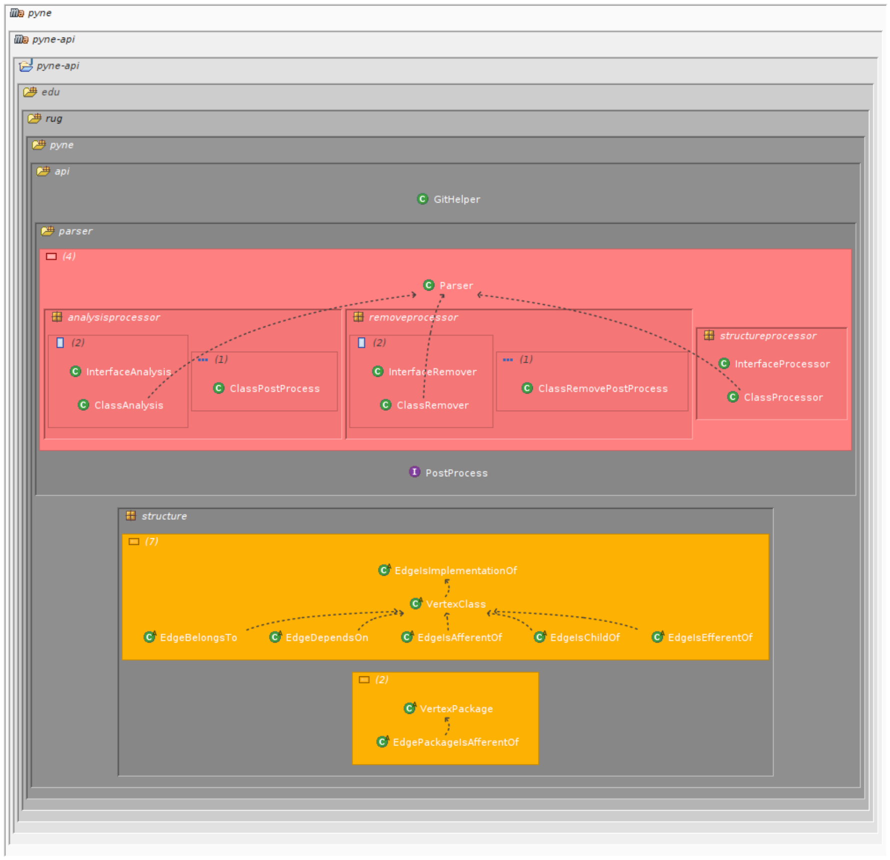

Figure 3.3: The graph of the relations between the classes of the pyne-api package created by Structure101

keep track of the files that have changed in the selected commit, and it holds lists of all types of processor classes, each of which are responsible for analysing the code in a different way.

The Parser class will then in turn communicate with the Analysis Processor package, Remove Processor package and Structure Processor package to create a dependency graph. These packages provide the following functionality: The Remove Processor package will take care of removing nodes or edges from the graph if classes or relations have been removed or changed. The Structure Processor package takes care of adding nodes to the dependency graph, and the Analysis Processor package analyses the different classes and packages and their relations to each other.

3.3 External dependencies

The main external dependencies used by the project are:

- **Apache Tinkerpop**

A flexible graphic framework. This library was chosen since Arcan, the software this project is supposed to mimic, also uses this framework.

- **Spoon**
  A Java source code parsing library. This library was chosen so that a Java parser did not need to be built from scratch [2].
- **Jgit**
  A Java library that provides Git functionality. This library was chosen so that the program is able to collect the source files using Git.
- **Apache Commons CLI**
  A Java library that provides CLI parsing and some basic CLI functionality. This library was used to not have to write a CLI parser from scratch.
- **log4j**
  A logging library that is used to log actions in the program.
- **Ferma**
  An extension to Apache Tinkerpop that is used to help building the graph.

Most of these dependencies were noted in the report [2], like Apacke Tinkerpop, Spoon and Jgit. Others, like Apache Commons Cli, log4j and Ferma, were found using Structure101.

In figure 3.5 and figure 3.4 you will find the graphs depicting the relation between the packages of Pyne and the dependencies. Note that the used external dependencies have not been shown for `pyne-demo`. This is because `pyne-demo` does not provide any graph creation functionality, only a "demo", and it uses approximately 20 extra external dependencies that are not used in the rest of the project. It would therefore be unreadable and not relevant to the working of Pyne itself.

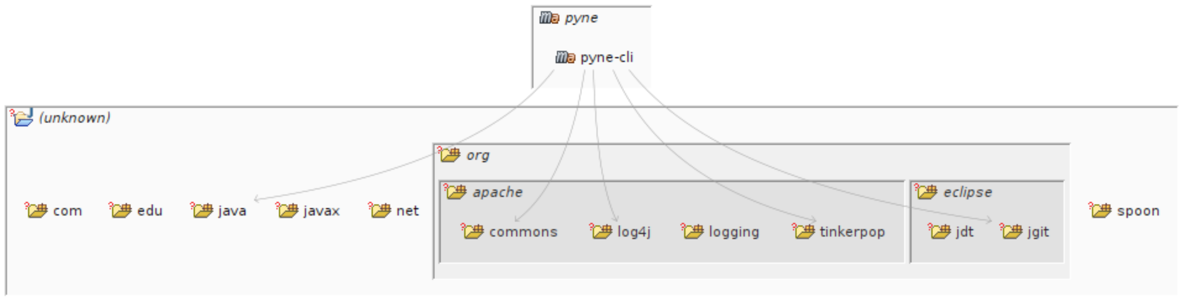

Figure 3.4: The graph of the external dependencies of the `pyne-cli` package, created by Structure101

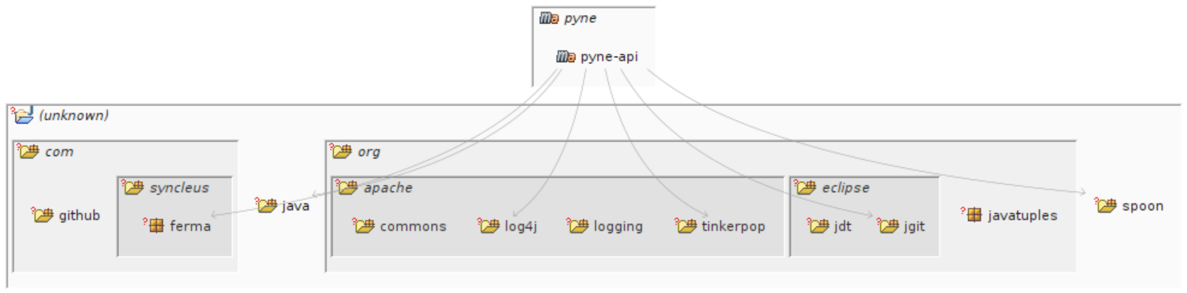

Figure 3.5: The graph of the external dependencies of the `pyne-api` package, created by Structure101

Chapter 4

Pyne versus Structure101

Contents

In this chapter...

In this chapter we will test both Pyne and Structure1010 on a large existing system. For this we have selected Apache Tajo (`www.tajo.apache.org`). The first section will describe the resulting graph for Tajo using Structure101. The following section will go into more detail about the design of Tajo using the graph generated by Structure101, then lastly we will run Pyne on Tajo and analyse the difference between the output of Pyne and that of Structure101. Note that in the commands we show in this chapter, we renamed the generated Pyne CLI jar to simply `pyne-cli.jar` for brevity.

4.1 Running Structure101 on Tajo

Next we will run Structure101 on Apache Tajo. Since Structure101 uses a GUI instead of a CLI, we followed the steps in the GUI to create a Structure101 project. For ease of use, we have provided this Structure101 project in our GitHub repository under `structure101/tajo_structure101.java.hsp`. As mentioned before, the downside of Structure101 is the need to compile the target system before it will allow you to import it. This was difficult since we had first switched to Java 11 to compile and run Pyne, but Tajo requires Java 8. Because this was not clearly indicated, it took some time to figure out.

Structure101 has more options for exporting as well, allowing users to export to PNG, JPEG, and "Graph as XML", which sounds a lot like graphml, but they are not the same, and in fact the program we used to open the graphml files did not display this file correctly.

Figure 4.1 shows what an exported image from Structure101 looks like.

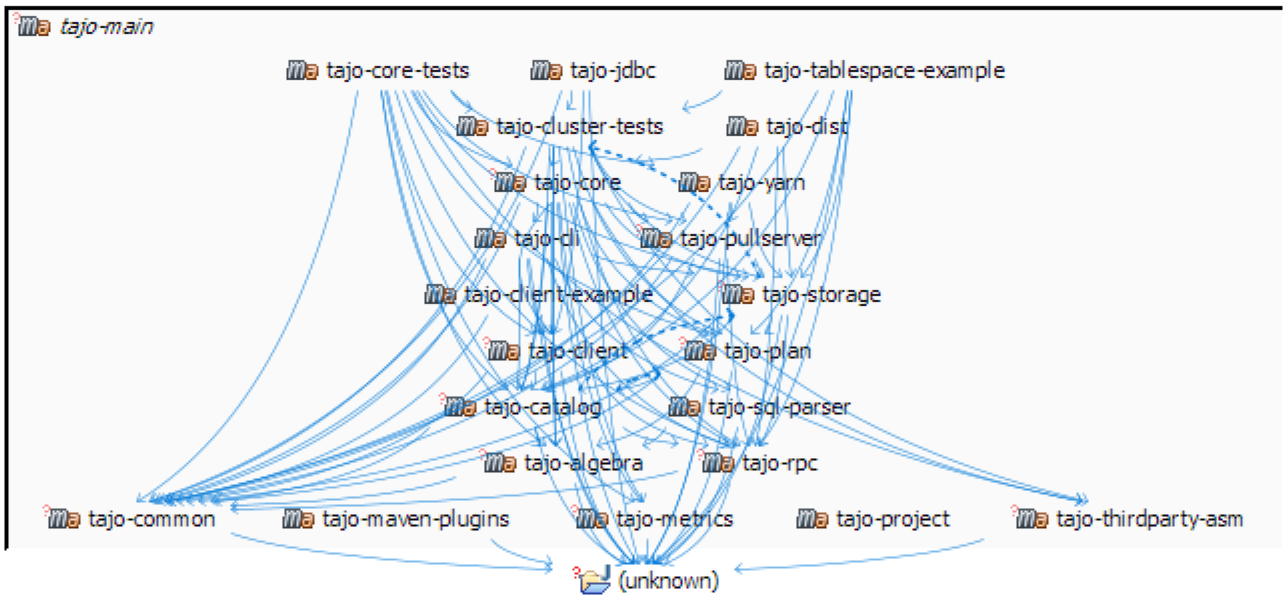

Figure 4.1: The graph of Tajo, created by Structure101

4.2 Design of Tajo

As you can see form figure 4.1, Tajo consists of several different packages working together. Firstly, We will briefly explain these components. Lastly we will go into more detail about the external dependencies Tajo relies on.

4.2.1 components

Here we will list the components of Apache Tajo, describe what they do and how they interact with each other. Below you will find a list of all the main components according to a maven build file in the source directory and Structure101. The responsibilities of each component has been approximated using the build file and the source code.

- `tajo-core` is responsible for the main functionality of Tajo. This component uses a master-worker pattern to perform its work. The master is responsible for query planning and coordinating the workers. It divides a query into sub-tasks and assigns these sub-tasks to individual workers. The workers are then responsible for executing these sub-tasks. The core structure can be seen in figure 4.2.

- `tajo-core-tests` and `tajo-cluster-tests` are responsible for testing the core and cluster respectively, where the focus is mainly on the correctness of query execution. Because they are testing libraries, they have only outgoing dependencies and are not really part of the core functionality. One exception, however, is the dependency of `tajo-storage` (more specifically the internal `tajo-storage-kafka` and `tajo-storage-pgsql` packages) on `tajo-cluster-tests`. This dependency, however, was caused by the fact that `tajo-cluster-tests` was added in as a dependency in the pom file, while none of the classes actually needed it. In other words, this dependency should have been removed from the pom file.

- `tajo-plan` is responsible for the planning or scheduling within Tajo. This package seems to be heavily used by the other components of the system. `tajo-core` for example has around 4500 references to `tajo-plan`.

- In the dependency graph of figure 4.1 You can see the `tajo-project` component and that it has no dependencies. That is because it is not a Java package. It is a parent POM for all Tajo Maven modules. All plugins and dependencies versions are defined here.

- `tajo-algebra` contains algebraic expressions for database interaction. An example of a couple of expressions here are `Join`, `DropTable`, `Sort` and `CreateDatabase`. It only depends on the `tajo-common` package within the project and has several other components, among which the `tajo-core` component, depending on it.

- `tajo-catalog` contains a catalog of plugins for a server, client, drivers and common. According to Structure101, this package seems to be used by a lot of components, and seems to be depending on a few too. One could therefore say that it is an important component.

- `tajo-common` contains some common modules. This consists of some Google Protobuf[6] definitions and some helper functions. Since this is basically a utility package, it does not depend on other components. Unsurprisingly, all the other major components do seem to depend on it.

- `tajo-client` is responsible for the client API and its implementation. The Tajo CLI will communicate with the Tajo Client, which will then in turn communicate with the server. The Tajo Client has a few minor incoming and outgoing dependencies, but the most notable one is that of the Tajo Core with 460 references. It goes without saying that the Tajo Client is an indispensable part of the system.

- `tajo-client-example` provides a client API example and `tajo-tablespace-example` provides a table example. No components seem to depend on these parts. This proves that the do not provide any core functionality, they just act as examples.
- `tajo-cli` provides the command line interface for the user to communicate with Tajo. It depends on a number of components in the system and even the Tajo Core seems to depend on it.
- `tajo-dist` This component is responsible for assembling the Tajo distribution. It depends on a few components, among which the Tajo Core, but no other component depends on it. Since this component should just assemble the Tajo distribution, no component should indeed depend on `tajo-dist`.
- `tajo-jdbc` represents the Tajo Java Database Connectivity driver. It is responsible for handling the communication with the database. `tajo-jdbc` depends on `tajo-common` and `tajo-client`, but no other class seems to depend on it. It could be that this package is no longer used by the system.
- `tajo-maven-plugins` contains the Maven plugins. From the dependency graph it can be seen that no component depends on `tajo-maven-plugins`. Therefore it is reasonable to assume that `tajo-maven-plugins` does not provide any functionality for Tajo, it only contains some plugins.
- `tajo-metrics` provides metrics for quantitative assessment of Tajo. Just like `tajo-algebra` it only provides a few functions for the Tajo Core and the two test classes to apply a metric. Therefore only those classes depend on it and it itself does not depend on anything else.
- `tajo-pullserver` is mainly responsible for fetching data from the server. The main components that use this service are `tajo-core` and `tajo-yarn`.
- `tajo-rpc` is responsible for the remote procedure calls. This components seems to be used by most of the components in the system. It is therefore also a key component in the system.
- `tajo-sql-parser` is, as the name implies, responsible for the parsing of SQL commands. This component is only used by the `tajo-cli` and `tajo-core`, but still very important.
- `tajo-storage` is responsible for storage and the relevant plugins. This component seems to have a lot of dependencies both incoming and outgoing. `tajo-core` even references it 530 times. It can therefore also be considered an important part of the system.
- `tajo-yarn` is presumably an extension of Tajo in order to use it with Yarn [7]. No main component seems to depend on Yarn, so we can assume it is not an significantly important component.
- `thirdpart-asm` is a Java code parser that provides functionality to parse compiled Java code. It seems to be only used by `tajo-core` and the test components.

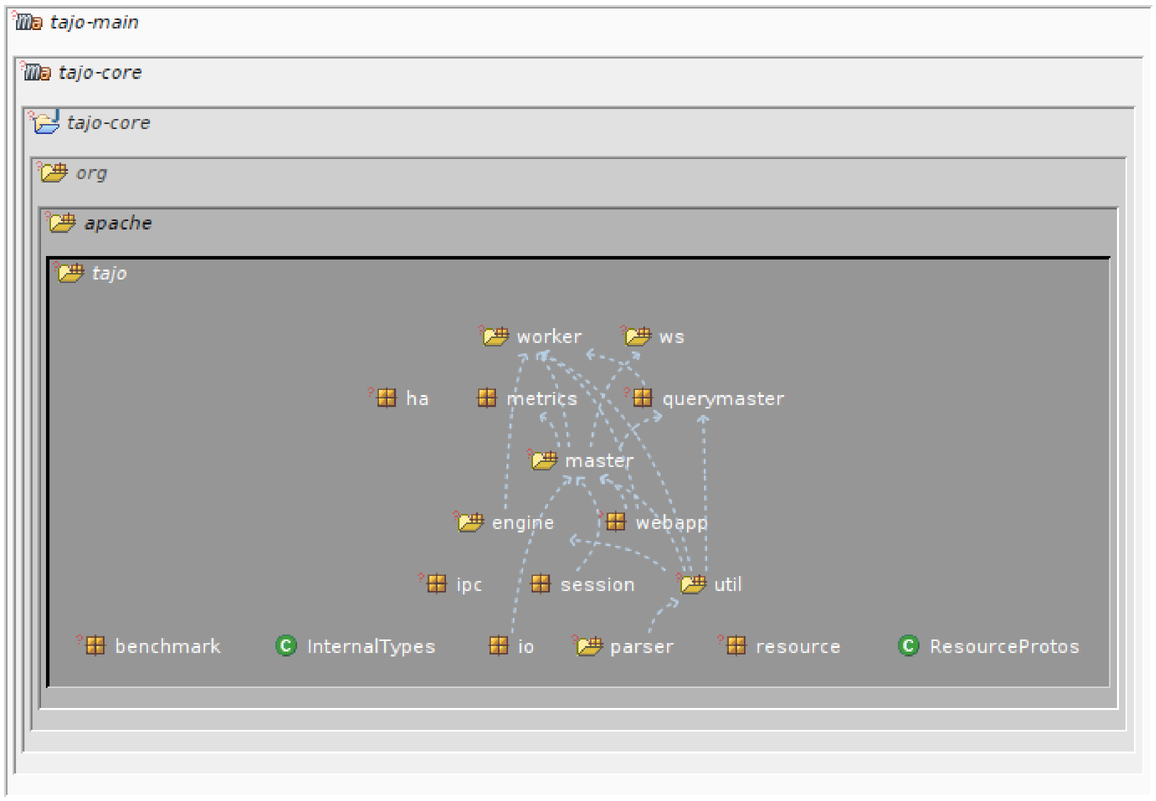

Figure 4.2: The graph of Tajo-core, created by Structure101

4.2.2 External dependencies

Here we will list a few of the main external dependencies of Tajo. This list will mainly include the external dependencies the `tajo-core` uses, since that means those libraries are part of the core functionality of Tajo. In addition to that, a few dependencies used by other components which can be considered important, will be listed as well.

One of the important external dependencies of Tajo is the Google Protocol Buffers [6] package. These are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data, much like JSON or XML, but smaller, faster and simpler. Tajo uses this to communicate with other services.

Another notable external dependency is that of Amazon AWS [8]. The features of this library are used by the `tajo-storage` component to enable authorisation or the enabling of retrieving credentials via the AWS service.

An important dependency used by the Tajo Core is Codahale [9] and gmetric4j [10]. These two library provide metrics to give insight into what the code does in production and measure the behavior of critical components in the production environment.

Another dependency of the Tajo Core is minidev.json [11], which provides functionality of reading and parsing JSON documents. The dependency jayway.jsonpath [12] is used in combination with minidev.json in a few minor cases to read the JSON documents.

Another remarkable dependency is the maxmind.geoip library [13]. This library is mainly used for identifying the location and other characteristics of an internet user. Looking at the source-code, it seems that Tajo is only after the country-code and then only uses that information to find out the correct timezone you're in. It looks like Tajo is not after people's personalised location data for ad revenue.

Two important dependencies for the Tajo CLI are the JLine library [14] and Jansi [15], which provides functionality to handle and format console input.

Antlr [16] is also used by both the Tajo Core and Tajo SQL Parser to generate a parser for reading, processing, executing or translating structured text or binary files. Looking at the source code of the Tajo Core, it seems to be only used for SQL parsing.

A very important dependency of the Tajo Core is Hadoop [17]. Not to mention that it is even required to have Hadoop installed in order to use Tajo. Hadoop provides a software framework for distributed storage and processing of big data.

Another important dependency used by the Tajo Core and `tajo-rpc` is glassfish.jersey [18]: An open source framework for developing RESTful web services in Java.

Snappy [19] is a dependency used by `tajo-common` for decompression/compression with the aim of high-speed and reasonable compression. This library is based of the original C++ version from Google, but now written in Java. According to the Git repository, it produces a byte-for-byte exact copy of the output created by the original C++ code.

The Tajo Core uses Mortbay Jetty [20] as a Java web server. More specifically, the Tajo Core uses this framework for the machine to machine communications.

The Tajo Core also uses reflection [21]. Reflection scans your classpath, indexes the meta-data, allows you to query it on runtime and may save and collect that information for many modules within your project.

Another neat dependency is the SLF4j [22], or the simple logging facade for Java. It allows the the end user to plug in the desired logging framework at deployment time.

4.3 Running Pyne on Tajo

We will start by running Pyne on Apache Tajo. The command we used to get a graph for the latest commit of Tajo is as follows:

```
java -jar pyne-cli.jar https://github.com/apache/tajo -s "2020-05-10" -e
  ↪ "2020-05-12" -p "DAY"
```

This creates a `.graphml` file[1], but unfortunately this does have some issues. See Appendix A.

[1]available on our github repository, under `graphs/tajo_dependencies_pyne.grahpml`

This is the only format Pyne can currently export, and opening a `.graphml` does require users to install a specialized program. We decided to use yEd Graph Editor[2] for this purpose. This has the advantage of being able to automatically rearrange the graph. Using this feature, we obtained figure 4.3.

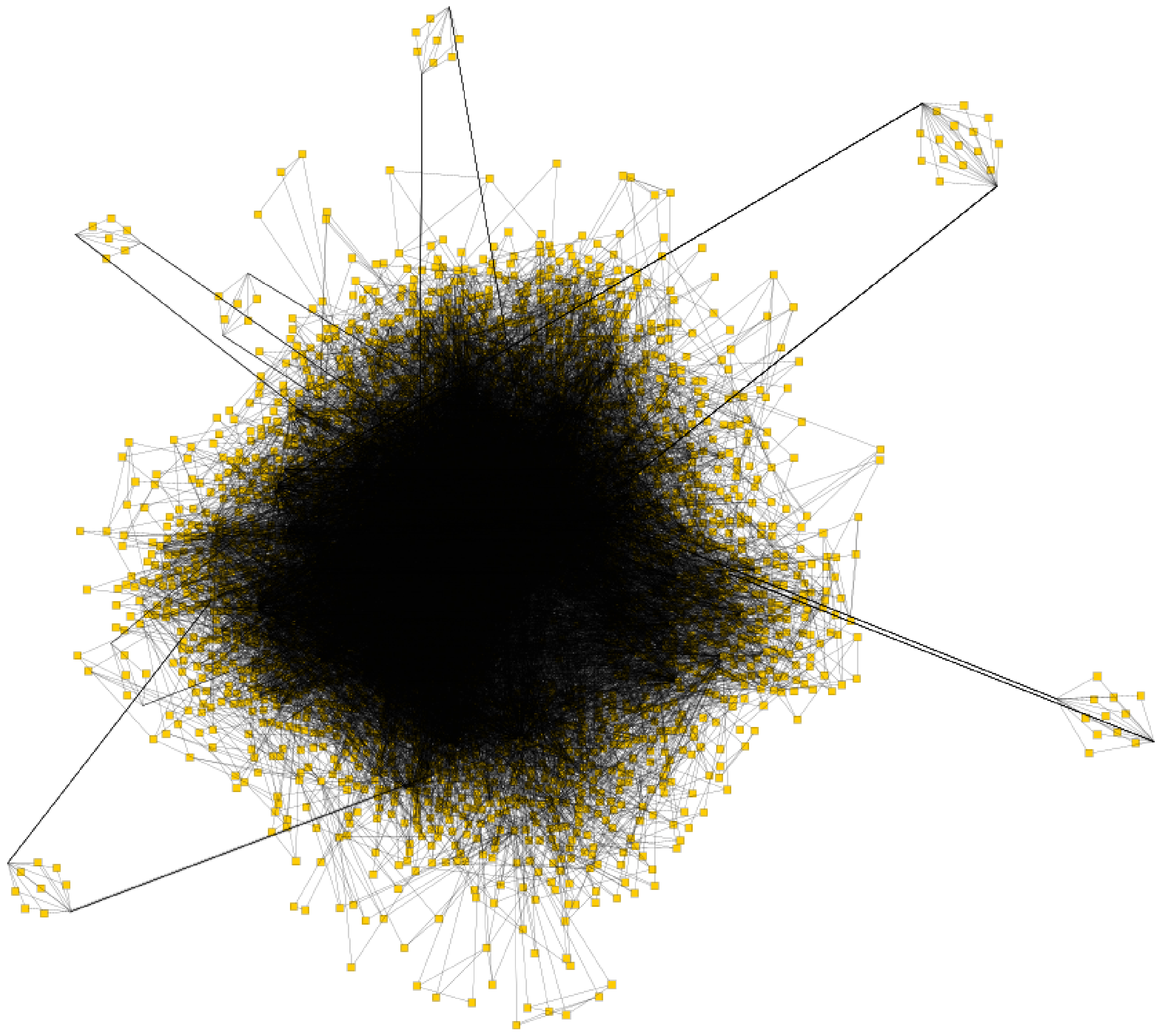

Figure 4.3: The automatically rearranged graph of Tajo, created by Pyne and rearranged by yEd

The nice thing about Pyne though, is that it can generate a dependency graph without the need to compile the target system first, which in the case of Tajo, proved to be a difficult task on its own as we will see later.

[2]`https://www.yworks.com/products/yed`

4.4 Comparing the feature sets

Feature	**Pyne**	**Structure101**
GUI	NO	YES
IDE plugin support	NO	YES[3]
Needs compiled sources	NO	YES
Shows external dependencies	YES	YES
Shows fully qualified class names	YES	NO[4]
Open-source	YES	NO
Freely available	YES	NO
Multiple OS support	YES	YES
Allows direct analysis of remote repositories	YES	NO
graphml export	YES	NO
PNG export	NO	YES
JPEG export	NO	YES
CSV export	NO	YES

Table 4.1: Comparing the features of Pyne and Structure101

4.5 Finding a way to compare the results of Pyne to Structure101's

To compare the results of Pyne with Structure101, we needed to obtain a machine-parsable format as the output of both tools. Pyne already does this, by exporting a graphml file, however for Structure101 this was a little more difficult to find, since Structure101 is mainly focused on exporting images. However, Structure101 does have the option to export to CSV and XML.

After exploring the exported CSV and XML from Structure101, we came to the conclusion that both formats have their own issues. On one hand, XML would be the preferable format, since an XML parser was already included in the dependency checker for the parsing of the graphml file from Pyne[5], but we found that Structure101's XML doesn't include the dependencies between the components, only the components themselves.

On the other hand, CSV does include the dependencies between the components, which makes this the more complete format to use. It has a different flaw, however, namely that it doesn't give the fully qualified class name (package name plus class name) for classes, but only the class name itself. This will make it difficult to compare the found classes, since we cannot compare whether they were found in the same package by both Pyne and Structure101.

[3] though this is quite a bit more limited than using the full program

[4] for some reason, it only show the fully qualified names for packages

[5] graphml is a standardised format to represent graphs that is built in XML, therefore it can be parsed by any XML parser

In the end, knowing the advantages and disadvantages of both formats, we decided to use CSV as the machine-parsable format that we needed, even though it is not an ideal file to work with.

Note that even though both of these programs are dependency graph creator engines, Pyne uses the source code to create the dependency graph, while Structure101 uses the jar files to create the dependency graph. This will cause unavoidable differences in output for both programs, which are not necessarily wrong.

Chapter 5

Comparing results programmatically

Contents

In this chapter...

In this chapter we will describe how we automated the comparison of the outputs of Pyne and Structure101

5.1 Creating a dependency checker program

Once we found a machine-parsable format we could export from both tools, we wrote a Java program[1] which extracts the found dependencies from both of the generated files. It then checks if there are any dependencies that were found by one tool, but not by the other, and provides some statistics on the results as well. A general flow diagram of this program can be seen in Figure 5.1.

5.1.1 Design of the dependency checker

The dependency checker was made with the technologies from Pyne kept in mind. This means we used the same Java version as Pyne does (version 11), and we export the results as an XML file. This way, not only does this mean that we don't have to awkwardly switch Java version each time we want to run one or the other, but it also means we can easily migrate the code from being an external tool to being a feature in Pyne, if we ever want to do so. In the same spirit, the structure of the program was very much designed to be modular. This was achieved by creating what is known as a black box around the internal functionalities, and exposing a set of intuitive methods that execute all steps of the comparison.

The program follows a simple and intuitive flow. First an instance of the `Comparator` class must be created. This is done by calling the constructor with the file names of the output files from Pyne and Structure101, together with some parameters that specify whether the analysis should be done on classes or packages, and a parameter that specifies the detail level the output should have.

Next, the file data should be imported into the program memory by using the `importFileData()` method. It takes no parameters, since the file names are already known, but it may throw exceptions related to the I/O regarding the files.

Then, using the `collectAllItems()` method, all items in the results from both Structure101 and Pyne can be converted into a general format that makes it possible to compare them.

To finish the comparison, the `compareResults()` method can be used. This method uses the general formatted items from the previous step to compare which tool(s) were able to find it. It returns the results in an XML tree that is ready to be output to a file.

Therefore the final step in the program is to use the `XMLHandler` class and its static method `writeXML()` to write the XML tree from the previous step to a file.

A flow diagram of this entire process can be found in Figure 5.1.

[1]Available in our GitHub repository, in the `dependency_checker/` folder

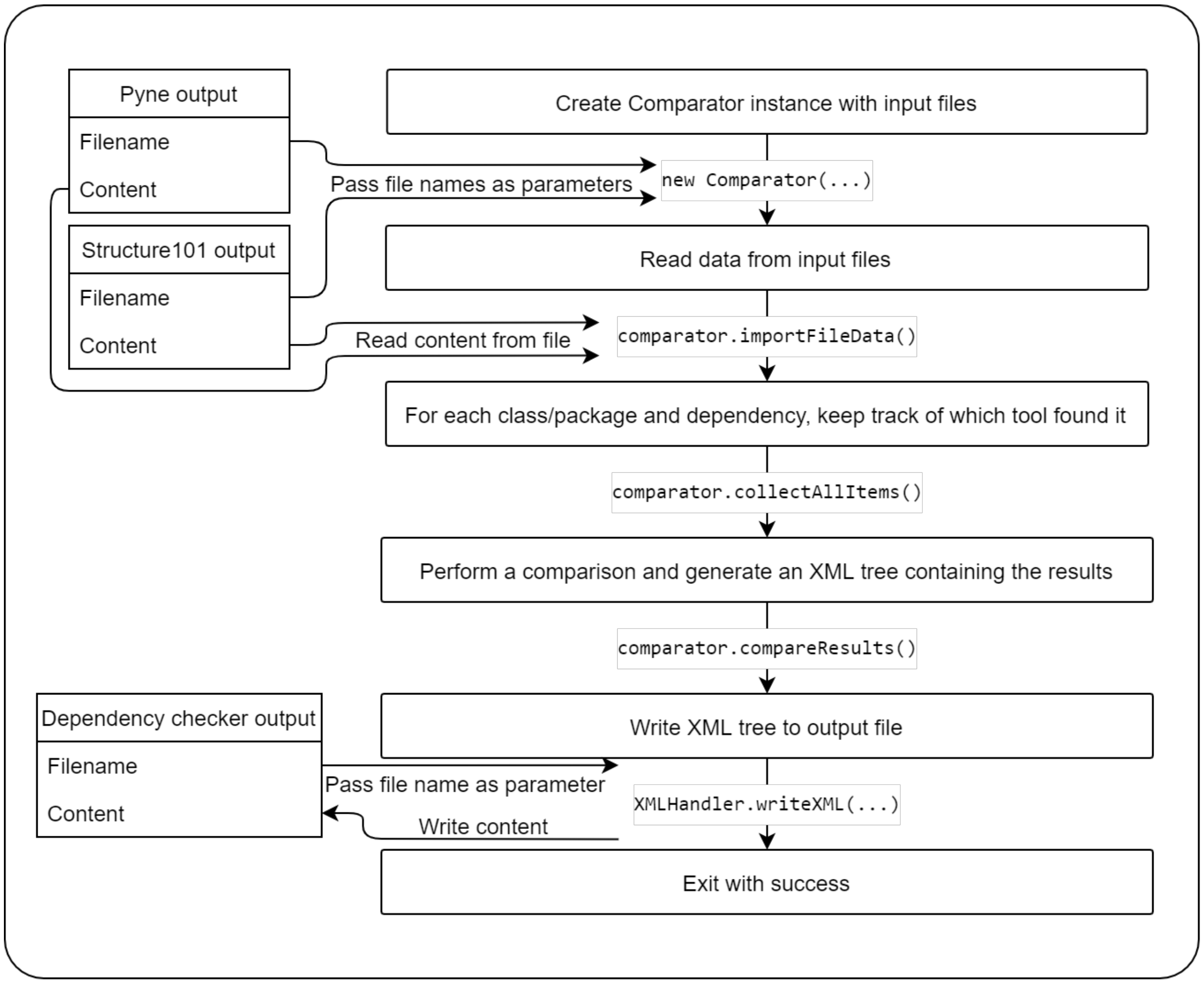
Pyne output
Filename
Content
Structure101 output
Filename
Content
Create Comparator instance with input files
Pass file names as parameters
new Comparator(...)
Read data from input files
Read content from file
comparator.importFileData()
For each class/package and dependency, keep track of which tool found it
comparator.collectAllItems()
Perform a comparison and generate an XML tree containing the results
comparator.compareResults()
Dependency checker output
Filename
Content
Write XML tree to output file
Pass file name as parameter
Write content
XMLHandler.writeXML(...)
Exit with success

Figure 5.1: The flow diagram of the dependency checker program

The structure of the XML output of our dependency checker can be found in Listing C.1, in Appendix C.

The root element is called ”results”, which has 3 children: ”allPackages”/”allClasses”, ”allDependencies” and ”tools”. The first contains the combined list of the packages or classes found by each individual tool. The second contains the combined list of the dependencies found by each individual tool. The last contains a list of all tools that we are comparing, where each tool contains:

- a list of packages or classes this tool did find
- a list of packages or classes this tool missed (by comparing the found packages or classes against the other tool’s found packages or classes)
- a list of the dependencies this tool did find
- a list of dependencies this tool missed (by comparing the found dependencies against the other tool’s dependencies)

Many elements also contain fields that provide some insights to the data without having to actually go through all items in the lists (which can get quite long for a big project, such as Apache Tajo). These fields include:

- "count", "countInternal", "countExternal", "countUnknown" tell the amount of items in a list, where "count" stands for the total amount of items, and the others tell the amount of items which are internal, external, or unknown, respectively.
- "internal" specifies whether a package is internal or not (i.e. external).
- "name" gives the name of the tool that the "tool" element represents.
- "percentageTotal", "percentageInternal", "percentageExternal", "percentageUnknown" specify the percentage that the amount of items in the list, which this field is on, is of all items of that category.

5.2 Running our dependency checker program

Using the fields on the XML elements of the output specified in subsection 5.1.1, we can obtain the statistics from comparing the unmodified version of Pyne to Structure101, using Apache Tajo as a comparison again. These can be found in Appendix C.2.

From that output we can see that Pyne initially performed much worse than Structure101, only finding less than half of the package Structure101 did, and less than two thirds of the classes. We also see that Pyne found one package that Structure101 didn't find though. After investigating, it turns out that the dependency that Structure101 missed was `PrimitiveType`. Our first instinct was that Pyne could use this to label primitive types like `int` or `bool`, but this turned out not to be true. When we investigated further, we found that this package contains one class according to Pyne, namely `PrimitiveTypeNameConverter`. Searching for this term online returns a JavaDoc page from Apache Parquet[2]. The reason why it didn't get labelled by its full name (`org.apache.parquet.schema.PrimitiveType.PrimitiveTypeNameConverter`) is unknown.

Furthermore, Pyne missed 100% of the unknown classes. This makes sense, since Structure101 very often doesn't mention for classes whether they are internal or external (it only does this consistently for packages). Pyne on the other hand can always tell, so it makes sense it would not have any classes marked as unknown.

[2] `https://www.javadoc.io/static/org.apache.parquet/parquet-column/1.8.2/org/apache/parquet/schema/PrimitiveType.html`

Chapter 6

Improving Pyne

Contents

In this chapter...

In this chapter we will go over the changes we made to Pyne in order for it to include all dependencies. We will be using the results from the dependency checker, which we discussed in chapter 5, and the source code of Pyne to find out what Pyne is missing and fix it. First of all, in section 6.1, we will list all the elements or "mistakes" in Pyne that make it output different or less dependencies than Structure101. Secondly, in section 6.2 we will list all these faults again and explain how we fixed them. Lastly, in section 6.3 we will compare the output of the dependency checker of the original version of Pyne with the improved version of Pyne (The version that has our implemented fixes). We will then evaluate to what extend our fixes have improved Pyne's ability to find dependencies.

6.1 Faults in Pyne

Here we will enumerate some of the "mistakes" in Pyne which makes its output different from Structure101. Note that the word mistake is a bit harsh here, some choices made in the implementation in Pyne are not necessarily wrong, but still give a different output. For example, Pyne does not include any folders with "test" or "example" in it. This is not necessarily wrong since those folders are not part of the main functionality of the program anyway. The first few points will discuss differences between Pyne and Structure101, which are not faults per se. The latter points will discuss the actual missing dependencies in Pyne and how it misses them. Most of these mistakes were made in the `ClassAnalysis` class, with most of them originating from the `getClassReferences()` function. The complete original code of Pyne can be seen in the respective GitHub repository [23]. We will explain how we fixed these issues in section 6.2.

1. **Files that are not in the source code are not parsed**
   This may be an obvious fault of Pyne, if you can even call it that, but it should be mentioned anyway. Since Pyne uses the source files to create the dependency graph and Structure101 uses the jar files, there are going to be unavoidable differences is the resulting dependency diagram. A lot of classes, mostly classes generated by Google Protobuf during compile time, are not in the source code yet and are only created when the program is compiled. It also goes the other way around. Some packages, like `tajo-docs` and `tajo-project` are not included in the jar file, while they are in the source code.

2. **Packages with no dependencies are skipped**
   This is a feature that is explicitly implemented in Pyne by the creator. Pyne has a whole post-process analysis where it traverses the graph and removes any nodes without edges. Structure101 does include packages without dependencies, so this feature will show a difference in output.

3. **Pyne does not parse any folder with "test" or "example" in its name or packages that do not have the folder "src" or "main" in it.**
   Pyne explicitly filters out any folders without a folder named "src" or "main" in it. It also removes any folders that contain the word "test" and "example". It is not wrong to consider example and test files as non-core functionality and use the "src" or "main" keyword to find project folders, but it does result in a different output that Structure101.

4. **Pyne does not check the constructor**
   Moving on to the actual faults in Pyne (or the missing dependencies), the first one we found was the missing check for the constructor in `getClassReferences()`. This function is responsible for returning the names (more specifically `CtTypeReferences`) of all classes this class depends on. It did go over each method in the class, but forgot about the constructor.

5. **Pyne uses the return type of an invocation for its dependency, not the declaration for the method itself.**

This was a mistake in the inner class `ExecutatbleConsumer`. The function `getClassReferences()` gave each invocation in the class, for example a method call like `dependencyClass.function()`, to the executable consumer which in turn should retrieve the name of the dependency. Instead of retrieving the class where the method was defined, it got the return type of the method. As can be seen in this code snippet:

```java
public void accept(CtElement element) {
    if (!(element instanceof CtExecutableReference<?>)) {
        return;
    }
    CtExecutableReference executable = (CtExecutableReference) element;
    try {
        CtTypeReference executableType = executable.getType();
        if (executableType != null && executableType.getDeclaration() != null) {
            dependences.add(executableType.getTypeDeclaration());
        }
    }catch (NullPointerException e){
        // Ignore Spoon errors
    }
}
```

If we ignore the empty catch statement made by the creator of Pyne (which is still painful to see) and focus our eyes on lines 5 and 7, we can see where it goes wrong. On line 5, the element is cast to an executable, which is the invocation that is being checked. Then on line 7, the type from this invocation is retrieved with `getType()`. This will however not return the declaring type, but the return type of the invocation. The function that should have been used is `getDeclaringType()`.

6. **Pyne had to retrieve the type declaration (the entire class + body) in order to make the dependency, but then later needed only the `referencedtype` (the full name).**
   This caused Spoon a lot of trouble when looking up external libraries. Getting the declaration was not necessary, only the name would suffice, so Spoon had to go back and forth for finding the reference name. For an example, see the code snippet below:

```java
private void processClassReferences(CtType clazz, VertexClass vertexClass) {
    for (CtType referencedClass : getClassReferences(clazz)) {
        if (referencedClass == null || referencedClass.getReference() == null) {
            continue;
        }
        VertexClass referencedClassVertex
                =
getOrCreateVertexClass(referencedClass.getReference());
            vertexClass.addDependOnClass(referencedClassVertex);
    }
}
```

```
private List<CtType> getClassReferences(CtType clazz) {...}
```

The `processClassReferences()` function takes a list of types (which is a definition for a clazz or interface). This list will then contain the class definition, methods and fields for each class this specific class depends on. But when creating the vertex on line 8, it only needs the reference, which is essentially just the package plus the name of the class. Since the type is retrieved from the reference in `getClassReferences()`, this step is completely unnecessary. `getClassReference` should return a `CtTypeReference`, not a `CtType`.

7. **Pyne did not look at the type of the parameters and return value of each method in the class.**
   There is no retrieval of parameters or return value for each method in `getClassReferences()`. Class A does depend on Class B if one of its methods uses class B as a parameter or return value, so it should be considered.

8. **Pyne did not look at the type of fields in each class.**
   Pyne does check the annotations of the fields, but not the type of the field. This means fields in a class are completely skipped.

9. **Spoon has some issues getting declarations of nested executable references.**
   This is actually not an issue with Pyne, but an issue with Spoon. To elaborate more on what we mean with "nested executable references", it is something like this: `dependencyClass.function().anotherFuction()`. Spoon has trouble finding out what the class is that defines `anotherFuction()`. This is especially a problem for external packages, where Spoon does not have the source code for either the class that defines `function()` or the class that defines `anotherFuction()`. Since this is a Spoon issue and not an issue with Pyne, we would need to replace the entire parser and rewrite Pyne almost entirely from scratch.

6.2 Changes to Pyne

Here we will go over each fault in Pyne and describe how we chose to fix it. Almost all of these fixes were performed in the `ClassAnalysis` class, of which the new source code can be found in our GitHub repository [24].

1. **Files that are not in the source code are not parsed**
   For obvious reasons, this issue could not be fixed. However to filter out all of the files that Structure101 finds but Pyne does not, we manually inspect all the missing internal dependencies of Pyne. From the resulting output we see that the only missing packages are not there because they are either not in the source code or do not have dependencies at all.

2. **Packages with no dependencies are skipped**
   Since this is also a core feature of Pyne, this issue could not be fixed. However we applied the same technique as before to inspect the missing internal dependencies of

Pyne. As mentioned, From the resulting output we see that the only missing packages are not there because they are either not in the source code or do not have dependencies at all.

3. **Pyne does not parse any folder with "test" or "example" in its name or packages that do not have the folder "src" or "main" in it.**
   We choose to include test and example folders in our project as well, since Structure101 does this too. This fix was as easy as removing the filters from the parser class. We did choose to keep the filter that specified that the folder must contain an "main" or "src" folder since Spoon would return errors if we removed this. The obvious problem is that Spoon cannot parse a directory that is not a source code directory.

4. **Pyne does not check the constructor**
   As you can see from the original code from Pyne [23], Pyne previously used this line of code to retrieve all of its methods.

```
for (CtMethod<?> ctMethod : (Set<CtMethod<?>>) clazz.getMethods())
```

   We added the constructor to this for loop with the following code:

```
ArrayList<CtExecutable<?>> executables = new ArrayList<>();
executables.addAll((Set<CtExecutable<?>>) clazz.getMethods());
if (clazz instanceof CtClass) {
    executables.addAll((Set<CtExecutable<?>>) ((CtClass)
 ↪ clazz).getConstructors());
}
for (CtExecutable<?> ctExecutable : executables)
```

   Essentially what we do is we add the list of methods and the list of constructors together and then loop over that instead. No real changes needed to be made to the body of the for loop, since most of the functions used there came from `CtExecutable` anyway, not from the child class `CtMethod`.

5. **Pyne uses the return type of an invocation for its dependency, not the declaration for the method itself.**
   On first sight, this looked like an easy fix. Just replace `getType()` with `getDeclaringType()`. However the `executableConsumer` also handled constructor calls (like `new dependencyClass()`) for which `getType()` was the correct method to use. That is why we decided to remove the `ExecutableConsumer` class and let `getClassReferences()` handle finding the references for constructor calls and invocations separately using the code below:

```
// Get all constructors in the method
List<CtConstructorCall<?>> constructorElements = body
        .getElements(new TypeFilter<>(CtConstructorCall.class));

//add all references for the constructor calls in the method
for(CtConstructorCall<?> c : constructorElements){
    references.add(c.getType());
}
```

```
// Get all invocations in the method
List<CtInvocation<?>> invocationElements = body
        .getElements(new TypeFilter<>(CtInvocation.class));

// Retrieve the dependencies of all invocations
for(CtInvocation<?> c : invocationElements){
    if(c.getExecutable().getDeclaringType() == null){
        LOGGER.warn("Spoon cannot find the declaration of " + c);
    }else {

  ↪ if(!(c.getExecutable().getDeclaringType().getTypeDeclaration()
  ↪ == null))
            references.add(c.getExecutable().getDeclaringType());
        else
            LOGGER.warn("Spoon cannot find the declaring type of " +
  ↪ c);
    }
}
```

6. **Pyne had to retrieve the type declaration (the entire class plus the body) in order to make the dependency, but then later needed only the `referencedtype` (the full name).**
This was a simple fix. The only thing that needed to be done was rewrite statements like `c.getType().getTypeDeclaration()` to `c.getType()` and change the return value of `getClassReferences()`.

7. **Pyne did not look at the type of the parameters and return value of each method in the class.**
To fix this a few extra checks needed to be added. This can be seen in the code below. This code snippet added the type of each method to the list of references.

```
references.add(ctExecutable.getType());
```

And this code snippet added all the annotations of the methods parameters and parameters types to the list of references.

```
for (CtParameter<?> parameter : ctExecutable.getParameters()) {
    parameter.getAnnotations().forEach(annotationConsumer);
    references.add(parameter.getType());
    tempCheck(parameter.getType(), clazz, parameter.toString());
}
```

8. **Pyne did not look at the type of fields in each class.**
To fix this a few extra checks needed to be added. This can be seen in the code below. This code snippet added all the annotations of the fields and field types to the list of references.

```
for (CtField<?> field : (List<CtField<?>>) clazz.getFields()) {
    field.getAnnotations().forEach(annotationConsumer);
    references.add(field.getType());
    tempCheck(field.getType(), clazz, field.toString());
}
```

9. **Spoon has some issues getting declarations of nested executable references.** This unfortunately could not be fixed. This is an issue that does not lie in Pyne, but in Spoon instead. The only way to fix this is use an entirely new parser and rewrite Pyne entirely. Even then it would not ensure that this problem will be fixed.

6.3 Result

6.3.1 Approach

Here we will compare the original version of Pyne to our improved version with the changes mentioned in section 6.2. Before we compared the two versions we first took the following steps:

1. we ran the original version of Pyne on Apache Tajo[4].
2. We compared the resulting graphml file to the CSV file from Structure101 using the dependency checker. The resulting output can be found in Appendix C.2.
3. We ran the improved version of Pyne on Apache Tajo[4].
4. We compared the resulting graphml file to the CSV file from Structure101 using the dependency checker. The resulting output can be found in Appendix C.3.
5. We put both outputs of the dependency checker in tables 6.1 and 6.2.
6. We calculated the recall, precision and F-measure for each table and each set of rows using the following formulas:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F = \frac{2 * Precision * Recall}{Precision + Recall}$$

where

$$TP = (\text{found } x \text{ by pyne} - \text{missed } x \text{ by structure101})$$

$$FP = \text{missed } x \text{ by structure101}$$

$$FN = \text{missed } x \text{ pyne}$$

and x can either stand for "packages", "classes", "package dependencies" or "class dependencies"

Using both table 6.1 and 6.2 we will try to compare the original version of Pyne to our improved version. This table contains information on the found and missed packages, found and missed package dependencies, found and missed classes and found and missed class dependencies of each program (Pyne and Structure101). In addition to that the precision, recall and F-measure are given for each of these group of rows.

The "found -" rows in the table represent the code elements (meaning packages, classes and dependencies) this particular program found. Note that in Pyne's case, this does not mean that these are the true code elements. Pyne might misidentify or misname a code element, resulting in a false positive. In structure101 case, this number represents the total existing code elements, since we assume Structure101's output to be the truth. The "missed -" rows in the table represent the code elements this particular program was not able to find, but the other was. So in the case of Pyne, one can perceive this number as the false negatives. In the case of Structure101, these are all the code elements Pyne mislabeled, or in other words, the false positives. So the correctly identified code elements can be found by subtracting the missed packages/classes/dependencies of Structure101 from the found packages/classes/dependencies of Pyne.

Note that the found/missed classes, class dependencies and corresponding precision, recall and f-measure are merely an indication and not the true value. This is because Structure101 does not provide a package name to the classes when extracting the graph to a CSV file, as mentioned in chapter 5, making the comparison to Pyne rather difficult. In reality, the missed classes and class dependencies might be lower and the found ones higher. On top of that, in our comparison we assume Structure101's results to be the ground truth, however, it is of course possible that Structure101 is flawed as well.

6.3.2 Analyses

Here we will analyse the table and actually compare the original version of Pyne to the improved version, while using the output of Structure101 as the ground truth. We will first note the difference in results of both table 6.1 and 6.2 and explain them. Secondly, we will compare the precision, recall and F-measure of each group of rows. Lastly, we will put everything together and conclude the analysis.

The first thing you can see is the increase in missed packages, classes and dependencies of Structure101 in table 6.2. This means that the improved version of Pyne finds packages, classes and dependencies that Structure101 did not find. This issue can be explained, however. Spoon sometimes has trouble retrieving the complete package name of (mostly external) dependencies, which makes it give only a subset of the package name. For example, if the actual package name was `package.external.project.classFile`, Spoon might only give `project.classFile`. In addition to that, most of the extra packages are `protoClasses` generated by Google Protobuf [6], which are generated during compile time and can produce issues with Spoon. However, while Pyne is able to find these dependencies, but the dependency checker only sees an incomplete package name, it will mark it as missing. Therefore, the number of missed package dependencies by Structure101 is in reality much lower.

	Pyne (original)	Structure101	Precision	Recall	F-score
Found packages	193	429	99.48%	44.76%	61.74%
Missed packages	237	1			
Found package dependencies	1020	3327	77.35%	23.72%	36.30%
Missed package dependencies	2538	231			
Found classes	2348	3259	69.72%	50.23%	58.39%
Missed classes	1622	711			
Found class dependencies	7809	36319	36.65%	7.88%	12.97%
Missed class dependencies	33457	4947			
		Average	**70.80%**	**31.65%**	**42.35%**

Table 6.1: Results of the dependency checker for the original version of Pyne

	Pyne (improved)	Structure101	Precision	Recall	F-score
Found packages	393	429	82.70%	75.76%	79.08%
Missed packages	104	68			
Found package dependencies	2431	3327	72.69%	53.11%	61.38%
Missed package dependencies	1560	664			
Found classes	383	3259	59.99%	70.64%	64.88%
Missed classes	957	1535			
Found class dependencies	28889	36319	23.03%	18.32%	20.40%
Missed class dependencies	29667	22237			
		Average	**59.60%**	**54.46%**	**56.43%**

Table 6.2: Results of the dependency checker for the improved version of Pyne

Looking at the missed package dependencies of Pyne, we see that there is an overall decrease in the improved version. This is a very good thing, since this means that the improved version of Pyne finds more of the classes found by Structure101 than the original version. This alone proves that our changes improved Pyne drastically.

The remaining missed dependencies, after manual inspection of around 30 of them, can be explained as follows. All of the inspected internal missing dependencies and a large portion of the external ones are caused by the fact that the classes are not in the source code, but are present in the jar file. This makes it virtually impossible for Spoon to find them. A lot of these classes are Google Protobuf [6] classes, which are presumably created during compile time.

One last issue explaining the rest of the inspected missing dependencies, is the fact that Spoon has trouble finding the declaration of nested dependencies. Since this is a Spoon specific issue, nothing can be changed to Pyne to fix it. Using another parser might be an option, but a full rewrite of Pyne will be needed and using a different parser might introduce new and different problems.

Looking at both table 6.1 and 6.2. We see that total amount of correctly identified packages by Pyne, calculated by subtracting the missed packages of Structure101 from the found packages of Pyne, increased from 192 to 325. This is an increase of **69.27%**.

Furthermore, the amount of correctly identified package dependencies by Pyne, calculated by subtracting the missed package dependencies of Structure101 from the found package dependencies of Pyne, increased from 789 to 1767. This is more than double the original dependencies.

Finally, we will be looking at the precision, recall and F-score of the different rows in both tables. We can see that the precision overall decreases somewhat with the changes we made, but the recall was much higher overall. The overall F-score, which incorporates both precision and recall, of both tables also sees an increase. This would also suggest that our changes did indeed improve Pyne.

To underline this even more, we can look at the average precision, recall and F-score of both tables. The original version of Pyne scored an average precision of **70.80%** , while the improved version showed an average precision of **59.60%**, which is a noticeable decrease. Looking at the recall we see that the original version of Pyne scored **31.65%** , while the improved version scored **54.46%** , which is significantly higher. Finally, the average of the F-score, which incorporates both precision and recall, has increased as well, with a **42.35%** score for the original version and **56.43%** for the improved version. So with these scores we can conclude that our improvements did increase the number of dependencies found by Pyne.

Chapter 7

Discussion

Contents

In this chapter...

In this chapter we will go over our process and discuss what we could have done differently. While we were successful in our pursuit to improve the performance of Pyne, it still doesn't perform quite as well as Structure101 does, as we saw in section 4.5. In the sections in this chapter we will describe the issues we encountered.

7.1 Diagnosing the reason for different outputs

The first issue we encountered during this project was that, while we were able to find the differences in found packages, classes, and dependencies between the two tools, it turned out to be quite difficult to diagnose the reason for these differences.

This is partially due to Structure101 giving very limited information in the files it exports, thus backtracing the differences to find any sort of pattern in the missing dependencies turned out quite difficult. This does make some sense because it is not the primary use case of Structure101, however it is a shame that this feature was implemented in such a basic manner while the U.I. of Structure101 seems much more feature rich. Still, in the end this only means that Structure101 might not have been the ideal tool to use for our use case.

7.2 Tajo's large size

Another issue we had while trying to diagnose the reason for the differences between Structure101 and Pyne is that Tajo is a very large project. This means that Tajo consists of lots of classes, packages, and dependencies. This made it difficult for us to verify the results of our dependency checker program during its development, and to manually find any sort of pattern in the items missing from the results of one tool or the other or spot bugs in the dependency checker program itself. The reason for this is that there could sometimes be thousands or even tens of thousands of items in a list, and manually backtracking those to the output of the other tools would be impossible due to the sheer amount. It was due to this reason also easier to try to find issues with Pyne in the source code of Pyne itself rather than its output, since the amount of items it reports is just too big to check manually.

Some possible ways to mitigate these issues would be to use a smaller comparison project instead of Tajo, maybe even one that is specifically meant to compare dependency graph creator engine programs, however finding one such project might be difficult in itself. Either way, a different comparison project might have made it easier for us to manually identify patterns in the results, since the amount of items in the list, and thus the amount of items we would have to check, would be smaller.

7.3 Development of the dependency checker

Finally, there are some issues with our dependency checker program. Like mentioned in section 6.3, the program currently has issues recognizing two packages as the same when one of the two doesn't specify the full name of the package, but only a subset. For example, consider a package `a.b.c.d.e`. If one tool lists it by its full name, `a.b.c.d.e`, and another lists it as `c.d.e`, they will currently not be detected as the same package. This could be fixed, but it may lead to false positives, so special care would have to be taken to ensure this detection would be as accurate as possible.

Chapter 8

Conclusion and Future work

Contents

In this chapter...

In this chapter we will conclude the document by going over the results and suggesting some future work. With that we also conclude the project.

8.1 Result recap

In conclusion, we achieved our goal of extending Pyne to discover more dependencies than it did before.

Looking at section 6.3 we saw an increase in found packages by the improved version of Pyne of nearly 70% compared to the original version. The amount of package dependencies even doubled compared to the original version.

In addition to that, looking at the average calculated precision, recall and F-measure from section 6.3, we saw very promising improvements as well when comparing the improved version with the original version of Pyne. Although the precision slightly decreased from 70% to roughly 60%, recall increased from 32% to 55%. Finally, the F-measure shows the most promise with an increase from 42% to 56%. So with these results we concluded that our changes to Pyne indeed improved the program quite a bit.

8.2 Future work

However, there is still room for improvement. In this section we will go into more detail about possible future work to Pyne or to this research.

8.2.1 Imperfections in Spoon

Firstly, Spoon seems to have a lot of trouble with finding the dependencies of nested invocations, as mentioned in section 6.3. Future work could include changing the Java source code parser used in Pyne from Spoon to something more mainstream like the Java parser library [25].

8.2.2 Adding method to method dependencies in Pyne

Another thing that might be a point of possible future improvement is method parsing. As already mentioned by the creators of Pyne [2], there is currently no functionality in Pyne that can add method to method dependencies to the graph. Since Structure101 does have this feature, it could be considered as a valuable option for Pyne as well.

8.2.3 Improve default CLI behavior of Pyne

As we already mentioned in Appendix A, the default CLI values of Pyne are quite confusing and illogical. The problem with this is that if the GitHub repository Pyne is analysing, does not have any commits between now and 5 days ago, no graph is created and no error is thrown. A better solution would be to set the default period to the latest commit, and only analyse that. This would solve a lot of confusion for first time users.

8.2.4 Perform more in depth investigation on Pyne

We saw that there was still a significant difference between structure101 and Pyne. Although research has show that no graph dependency checker tool gives exactly the same output [1], which makes bench-marking nearly impossible, we could try to apply Pyne to other programs than Tajo in order to find other faults and improve it. Tajo was quite a big program and we might have missed some ”missing” dependencies in Pyne. Applying Pyne to something smaller like Apache Tika [26], might make it easier to find other existing problems within Pyne.

References

[1] Leo Pruijt et al. "The accuracy of dependency analysis in static architecture compliance checking: The Accuracy of Dependency Analysis in Static ACC". In: *Software: Practice and Experience* 47 (July 2016). DOI: `10.1002/spe.2421`.

[2] Patrick Beuks. *Building a dependency graph from Java source code files*. 2019.

[3] Chris Chedgey and Paul Hickey. *structure101*. Version V4.2 b15352. Dec. 14, 2020. URL: `https://structure101.com/`.

[4] Apache Software Foundation. *Tajo*. Version 0.12.0-SNAPSHOT. Dec. 14, 2020. URL: `https://tajo.apache.org/`.

[5] HeadwaySoftware. *XS – A Measure of Structural Over-Complexity*. 2006. URL: `https://structure101.com/static-content/pages/resources/documents/XS-MeasurementFramework.pdf`.

[6] Google. *com.google.protobuf*. Dec. 14, 2020. URL: `https://developers.google.com/protocol-buffers`.

[7] *yarn*. Dec. 14, 2020. URL: `https://yarnpkg.com/`.

[8] Amazon. *com.amazonaws*. Dec. 14, 2020. URL: `https://aws.amazon.com/`.

[9] Dropwizard. *io.dropwizard.metrics*. Dec. 14, 2020. URL: `https://metrics.dropwizard.io`.

[10] ganglia. *info.ganglia.gmetric4j*. Version 1.0.3. Dec. 14, 2020. URL: `https://github.com/ganglia/gmetric4j`.

[11] Minidev. *net.minidev*. Dec. 14, 2020. URL: `https://mvnrepository.com/artifact/net.minidev/json-smart`.

[12] Jayway. *com.jayway.jsonpath*. Dec. 14, 2020. URL: `https://github.com/json-path/JsonPath`.

[13] Maxmind. *com.maxmind.geoip*. Version 1.2.15. Dec. 14, 2020. URL: `https://www.maxmind.com/en/geoip2-services-and-databases`.

[14] Maxmind. *jline*. Dec. 14, 2020. URL: `https://github.com/jline/jline3`.

[15] Fusesource. *org.fusesource.leveldbjni*. Version 1.8. Dec. 14, 2020. URL: `https://github.com/fusesource/jansi`.

[16] Terence Parr. *antlr*. Dec. 14, 2020. URL: `https://www.antlr.org/`.

[17] Apache Software Foundation. *Hadoop*. Version 2.3.0+. Dec. 14, 2020. URL: `https://hadoop.apache.org`.

[18] glassfish. *glassfish.jersey*. Dec. 14, 2020. URL: `https://eclipse-ee4j.github.io/jersey/`.

[19] dain. *org.iq80.snappy*. Dec. 14, 2020. URL: `https://github.com/dain/snappy`.

[20] Mortbay. *org.mortbay.jetty*. Dec. 14, 2020. URL: https://mvnrepository.com/artifact/org.mortbay.jetty.

[21] ronmamo. *org.reflections*. Dec. 14, 2020. URL: https://github.com/ronmamo/reflections.

[22] qos-ch. *slf4j*. Dec. 14, 2020. URL: http://www.slf4j.org/.

[23] darius-sas. *Pyne*. Version 1.1-SNAPSHOT. Jan. 9, 2021. URL: https://github.com/darius-sas/pyne.

[24] Job Heersink, Richard Westerhof. *Our Github Repository*. Jan. 9, 2021. URL: https://github.com/richardswesterhof/pyne.

[25] Danny van Bruggen. *javaparser*. Version 3.18.0. Jan. 10, 2021. URL: https://javaparser.org/.

[26] Apache. *Tika*. Version 1.24.1. Jan. 10, 2021. URL: https://tika.apache.org/.

Appendix A

Troubleshooting

Here we present several problems encountered and how we tried to solve them.

1. **Bug in graph creator**:
   When a graph is opened in a graph editor, like draw.io or yEd graph editor, only one square appears, but the file is 2.25 MB, which implies there is much more to it. It turned out that all the squares are stacked upon each other, but the relation between the squares is still there (which is visible from the neighbours tab in yEd graph editor).
   It seems that the squares themselves do not contain any data on the classes they represent. They do not have any value assigned to them. Furthermore, opening any graph will give a warning about the 'linesOfCode' property being of type long, which apparently isn't supported.

2. **Default CLI values**:
   The default options for the CLI are not the best choice. Namely the period option is set to days by default, which will create a separate graph for each day. The start date is set to 5 periods (so 5 days on default) from the end date (which is the current date by default). It is relatively rare for a git repository to have commits from the past 5 days.

3. **Error handling**:
   In a few cases where something goes wrong or no graph is created, no error is thrown and the user does not know why no graph has been made. For that reason more error logging statements need to be added.

4. **Java Version**:
   Pyne requires Java version 11, however to compile Tajo, Java 8 is needed. This leads to confusing errors if you try to compile Tajo for yourself after using Java 11 to compile Pyne.

5. **First manual inspection**:
   On the first manual inspection of the dependencies Pyne is not able to see using

the source code and our created dependency checker, the following mistakes were recognized:
Pyne is not able to inspect the constructor. (Adding this fixed two packages)
Pyne is not able to inspect arguments of methods and constructors.
Pyne is not able to inspect return values of methods.

6. **Scanned directories**:
   After manually analysing the directories from the git repository added to Pyne by the GitHelper, we already found some inconsistencies. `tajo-core-test`, `tajo-tablespace-example`, `tajo-cluster-tests`, `tajo-client-example`, `tajo-project` are all missing from the list of added directories. In addition to that `tajo-docs` is present in Pyne, while not present in Structure101. The latter is probably due to the fact that `tajo-docs` is not present in the jar-file, but only in the Git repository. Looking further into the source code of Pyne, it seems that in the `findSourceDirectories()` method, Pyne specifically filters out any directories with ”test” or ”example” in it and any directory that does not contain a ”src/java” or ”src/main” in their directory. Since there is no real reason why tests should not be tested on dependencies, this filter should be removed.

7. **Missing packages in graph**:
   The missing packages in the graph are probably mainly due to the fact that Pyne filters out packages without incoming or outgoing dependencies.

8. **Inconsistencies Structure101 and source code**:
   Since Structure101 analyses the jar files and Pyne the source files, there are going to be some obvious inconsistencies. When comparing the source code structure with the Structure101 diagram, you might notice that a few minor packages within the components are missing from the source code or missing from the diagram. We will have to keep this in mind.

9. **Compiling Tajo**:
   To let Structure101 analyse Tajo, we had to compile it first. While the requirements for compiling Tajo[1] say that Java 8 or above can be used, we were not able to compile it with Java 11, so we used Java 8 instead. On top of that, it seems like there is one failing test case during the build stage, which prevents the build from finishing. To get around this, we compiled with the `-DskipTests` flag in Maven. This might be fixed in later versions of Tajo.

[1]`https://github.com/apache/tajo#requirements`

Appendix B

Change Log

ID	Author	Student Number	Email Address
JH	Job Heersink	s3364321	j.g.heersink@student.rug.nl
RW	Richard Westerhof	s3479692	r.s.westerhof.2@student.rug.nl

Table B.1: The authors that contributed to this document

Ver.	ID	Date	Revision
0.1	JH	21-11-2020	Create User guide.
	JH	21-11-2020	Create Design section.
	JH	21-11-2020	Create problem (troubleshooting) section.
	RW	22-11-2020	Added classes and dependency graph to design
	RW	22-11-2020	Revised User guide
	RW	22-11-2020	Revised problem section

Ver.	ID	Date	Revision
0.2	JH	01-12-2020	Reformatted the document.
	JH	01-12-2020	Added frontpage.
	JH	01-12-2020	Added introduction.
	JH	01-12-2020	Rewrote design section.
	JH	01-12-2020	Moved and renamed troubleshooting section to appendix.
	RW	01-12-2020	Create chapter 4.
	RW	01-12-2020	Add ”java version” item to Troubleshooting appendix.

Ver.	ID	Date	Revision
0.3	RW	03-12-2020	Incorporate TA feedback in document.
	JH	04-12-2020	Added external dependencies section.
	JH	05-12-2020	Added requirements section.
	RW	07-12-2020	Improve internal structure of document and improve consistency.
	RW	08-12-2020	Add XS diagram and graphs of Tajo created by both Structure101 and Pyne.

Ver.	ID	Date	Revision
0.4	JH	12-12-2020	Incorporate TA feedback in document.
	JH	12-12-2020	Add design of Tajo.
	JH	13-12-2020	Add external dependencies of Tajo.
	JH	14-12-2020	Added all software to reference list.
	RW	14-12-2020	Write code for automatic comparison, perform analysis using results from this comparison.
	RW	14-12-2020	Write section 4.5 (reporting on code and analysis).

Ver.	ID	Date	Revision
0.5	RW	21-12-20	Expanding section Creating a dependency checker program.
	RW	24-12-20	Write more details in Creating a dependency checker program.
	JH	29-12-20	Created chapter 6 and added faults of pyne.
	JH	30-12-20	Added improvements made to Pyne in chapter 6
	RW	02-01-21	Moved dependency checker sections to new chapter.
	RW	02-01-21	Added flow diagram of dependency checker.
	JH	02-01-21	Added comparison between original and improved Pyne in chapter 6.
	JH	03-01-21	Added explanation to figure 3.1.
	JH	03-01-21	Added section on running the dependency checker.
	RW	04-01-21	Added Tajo compilation to issues section.
	RW	04-01-21	Wrote user guide section for Structure101.
	RW	04-01-21	Expanded user guide section for dependency checker.
	JH	05-01-21	Updated introduction section.
	RW	05-01-21	Fixed spelling in the document.
	RW	05-01-21	Added results for class level analyses.

Ver.	ID	Date	Revision
0.6	RW	08-01-2021	Created discussion and conclusion chapters.
	JH	09-01-2021	Updated chapter 6.
	JH	09-01-2021	Updated appendix C.
	JH	10-01-2021	Updated chapter 8.
	RW	12-01-2021	Some spelling and grammar fixes.

Ver.	ID	Date	Revision
0.7	RW	14-01-2021	Created subsections in discussion and conclusion chapters, and a few small additions to those chapters as well.
	RW	14-01-2021	Proof reading entire document, fixing most spelling, grammar and style errors.
	JH	14-01-2021	Fixed chapter 6
	JH	14-01-2021	Fixed chapter 1
	RW	16-01-2021	Added minitocs and chapter introductions.
	RW	16-01-2021	Added description of the dependency checker flow diagram (Figure 5.1).

Appendix C

Dependency checker output

This chapter contains all the output of the dependency checker program. Most of these outputs, expect for the template, will contain information on the comparison of the dependencies found by structure101 to the dependencies found by some version of Pyne. Note that in the original xml output files contain the individual missing packages, classes and dependencies as well. Those have been left out to avoid cluttering the document.

C.1 Output Template

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<results>
  <allPackages count="Integer" countExternal="Integer"
  ↪ countInternal="Integer" countUnknown="Integer">
    <pkg id="Integer" internal="Boolean">name.of.package</dependency>
    etc.
  </alPackages>
  <allDependencies count="Integer">
    <dep fromID="Integer" toID="Integer">
      <!-- the following elements are only present when the
  ↪ "human-readable" option is used -->
      <fromIsInternal>Boolean</fromIsInternal>
      <fromName>name.of.package</fromName>
      <toIsInternal>Boolean</toIsInternal>
      <toName>name.of.other.package</toName>
    </dep>
    etc.
  </allDependencies>
  <tools count="Integer">
    <tool name="String">
      <foundPackages count="Integer" countExternal="Integer"
  ↪ countInternal="Integer" countUnknown="Integer"
  ↪ percentageExternal="Float" percentageInternal="Float"
  ↪ percentageTotal="Float" percentageUnknown="Float">
        <!-- the "internal" attribute and the package name are not
  ↪ present when the "compact" option is used -->
```

```
        <pkg id="Integer" internal="Boolean">name.of.package</pkg>
        etc.
      </foundPackages>
      <missedPackages count="Integer" countExternal="Integer" countInternal="Integer" countUnknown="Integer" percentageExternal="Float" percentageInternal="Float" percentageTotal="Float" percentageUnknown="Float">
        <!-- the "internal" attribute and the package name are not present when the "compact" option is used -->
        <pkg id="Integer" internal="Boolean">name.of.package</pkg>
        etc.
      </missedPackages>
      <foundDependencies count="Integer" percentageTotal="Float">
        <dep fromID="Integer" toID="Integer">
          <!-- the following elements are only present when the "human-readable" option is used -->
          <fromIsInternal>Boolean</fromIsInternal>
          <fromName>name.of.package</fromName>
          <toIsInternal>Boolean</toIsInternal>
          <toName>name.of.other.package</toName>
        </dep>
        etc.
      </foundDependencies>
      <missedDependencies count="Integer" percentageTotal="Float">
        <dep fromID="Integer" toID="Integer">
          <!-- the following elements are only present when the "human-readable" option is used -->
          <fromIsInternal>Boolean</fromIsInternal>
          <fromName>name.of.package</fromName>
          <toIsInternal>Boolean</toIsInternal>
          <toName>name.of.other.package</toName>
        </dep>
        etc.
      </missedDependencies>
    </tool>
    etc.
  </tools>
</results>
```

Listing C.1: The structure of the output of our dependency checker

C.2 ORIGINAL VERSION OF PYNE

C.2.1 CLASSES

```xml
<results>
    <allClasses count="3970" countExternal="237" countInternal="2184"
  ↪ countUnknown="1549"></allClasses>
    <allDependencies count="41266"></allDependencies>
    <tools count="2">
        <tool name="STRUCTURE101">
            <foundClasses count="3259" countExternal="223"
  ↪ countInternal="1487" countUnknown="1549"
  ↪ percentageExternal="94.09283" percentageInternal="68.08608"
  ↪ percentageTotal="82.09068" percentageUnknown="100.0"></foundClasses>
            <missedClasses count="711" countExternal="14"
  ↪ countInternal="697" countUnknown="0" percentageExternal="5.907173"
  ↪ percentageInternal="31.913918" percentageTotal="17.909319"
  ↪ percentageUnknown="0"></missedClasses>
            <foundDependencies count="36319"
  ↪ percentageTotal="88.01192"></foundDependencies>
            <missedDependencies count="4947"
  ↪ percentageTotal="11.988077"></missedDependencies>
        </tool>
        <tool name="PYNE">
            <foundClasses count="2348" countExternal="164"
  ↪ countInternal="2184" countUnknown="0" percentageExternal="69.19831"
  ↪ percentageInternal="100.0" percentageTotal="59.143578"
  ↪ percentageUnknown="0"></foundClasses>
            <missedClasses count="1622" countExternal="73"
  ↪ countInternal="0" countUnknown="1549" percentageExternal="30.801687"
  ↪ percentageInternal="0" percentageTotal="40.856422"
  ↪ percentageUnknown="100.0"></missedClasses>
            <foundDependencies count="7809"
  ↪ percentageTotal="18.923569"></foundDependencies>
            <missedDependencies count="33457"
  ↪ percentageTotal="81.07643"></missedDependencies>
    </tool>
  </tools>
</results>
```

Listing C.2: A comparison of results on a class level before modifying Pyne

C.2.2 PACKAGES

```
<results>
    <allPackages count="430" countExternal="242" countInternal="188"
  ↪ countUnknown="0"></allPackages>
    <allDependencies count="3558"></allDependencies>
    <tools count="2">
        <tool name="STRUCTURE101">
            <foundPackages count="429" countExternal="241"
  ↪ countInternal="188" countUnknown="0" percentageExternal="99.58678"
  ↪ percentageInternal="100.0" percentageTotal="99.76744"
  ↪ percentageUnknown="0"></foundPackages>
            <missedPackages count="1" countExternal="1" countInternal="0"
  ↪ countUnknown="0" percentageExternal="0.41322312"
  ↪ percentageInternal="0" percentageTotal="0.23255813"
  ↪ percentageUnknown="0"></missedPackages>
            <foundDependencies count="3327"
  ↪ percentageTotal="93.50759"></foundDependencies>
            <missedDependencies count="231"
  ↪ percentageTotal="6.492411"></missedDependencies>
        </tool>
        <tool name="PYNE">
            <foundPackages count="193" countExternal="47"
  ↪ countInternal="146" countUnknown="0" percentageExternal="19.421488"
  ↪ percentageInternal="77.65958" percentageTotal="44.88372"
  ↪ percentageUnknown="0"></foundPackages>
            <missedPackages count="237" countExternal="195"
  ↪ countInternal="42" countUnknown="0" percentageExternal="80.578514"
  ↪ percentageInternal="22.340425" percentageTotal="55.11628"
  ↪ percentageUnknown="0"></missedPackages>
            <foundDependencies count="1020"
  ↪ percentageTotal="28.66779"></foundDependencies>
            <missedDependencies count="2538"
  ↪ percentageTotal="71.33221"></missedDependencies>
        </tool>
    </tools>
</results>
```

Listing C.3: A comparison of results on a package level before modifying Pyne

C.3 IMPROVED VERSION OF PYNE

C.3.1 CLASSES

```xml
<results>
    <allClasses count="4794" countExternal="1620" countInternal="2277" countUnknown="897">
    <allDependencies count="58556">
    <tools count="2">
        <tool name="STRUCTURE101">
            <foundClasses count="3259" countExternal="859" countInternal="1503" countUnknown="897" percentageExternal="53.024693" percentageInternal="66.007904" percentageTotal="67.980804" percentageUnknown="100.0">
            <missedClasses count="1535" countExternal="761" countInternal="774" countUnknown="0" percentageExternal="46.975307" percentageInternal="33.992092" percentageTotal="32.01919" percentageUnknown="0">
            <foundDependencies count="36319" percentageTotal="62.024384">
            <missedDependencies count="22237" percentageTotal="37.975613">
    </tool>
        <tool name="PYNE">
            <foundClasses count="3837" countExternal="1560" countInternal="2277" countUnknown="0" percentageExternal="96.296295" percentageInternal="100.0" percentageTotal="80.037544" percentageUnknown="0">
            <missedClasses count="957" countExternal="60" countInternal="0" countUnknown="897" percentageExternal="3.7037036" percentageInternal="0" percentageTotal="19.962452" percentageUnknown="100.0">
            <foundDependencies count="28889" percentageTotal="49.33568">
            <missedDependencies count="29667" percentageTotal="50.66432">
    </tool>
  </tools>
</results>
```

Listing C.4: A comparison of results on a class level after modifying Pyne

C.3.2 PACKAGES

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<results>
    <allPackages count="497" countExternal="305" countInternal="192"
  ↪ countUnknown="0">
    <allDependencies count="3991">
    <tools count="2">
        <tool name="STRUCTURE101">
            <foundPackages count="429" countExternal="237"
  ↪ countInternal="192" countUnknown="0" percentageExternal="77.70492"
  ↪ percentageInternal="100.0" percentageTotal="86.31791"
  ↪ percentageUnknown="0">
            <missedPackages count="68" countExternal="68"
  ↪ countInternal="0" countUnknown="0" percentageExternal="22.295082"
  ↪ percentageInternal="0" percentageTotal="13.682093"
  ↪ percentageUnknown="0">
            <foundDependencies count="3327" percentageTotal="83.362564">
            <missedDependencies count="664" percentageTotal="16.637434">
        </tool>
        <tool name="PYNE">
            <foundPackages count="393" countExternal="236"
  ↪ countInternal="157" countUnknown="0" percentageExternal="77.37705"
  ↪ percentageInternal="81.77083" percentageTotal="79.07445"
  ↪ percentageUnknown="0">
            <missedPackages count="104" countExternal="69"
  ↪ countInternal="35" countUnknown="0" percentageExternal="22.62295"
  ↪ percentageInternal="18.229168" percentageTotal="20.925552"
  ↪ percentageUnknown="0">
            <foundDependencies count="2431" percentageTotal="60.912056">
            <missedDependencies count="1560" percentageTotal="39.087948">
        </tool>
    </tools>
</results>
```

Listing C.5: A comparison of results on a package level after modifying Pyne