university of groningen

faculty of science and engineering

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

December 15, 2020

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.

In this document we will compare an incomplete existing dependency graph creator called Pyne[1] with another widely used dependency graph creator engine called Structure101[2], which we will consider as complete. In addition to that we will find the missing functionality in Pyne, which Structure101 does offer, and implement this functionality in Pyne.

The document is structured as follows: Chapter 2 will show how Pyne should be installed and used. Chapter 3 will discuss the design of Pyne itself. In Chapter 4 we will apply Pyne and Structure101 to a large java application called Apache Tajo[3] and finally in Chapter 5 and 6 we will discuss and conclude the project.

Chapter 2

User Guide

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, one can use this application 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"
```

Chapter 3

Design

Here we will go into more detail about the design of Pyne itself. The project has been analysed using the existing documentation [1], the source code and the structural and dependency graphs created by Structure101 (figure 3.1).

3.1 Requirements

According to the documentation [1] of the Pyne project, the project has the following requirements:

1. **The program should be able to create a graph from Java source files using git.**
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 needs to be able to use 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,

[1] `https://essere.disco.unimib.it/wiki/arcan/`
[2] `https://tinkerpop.apache.org/`
[3] `http://spoon.gforge.inria.fr/`

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. Furthermore, Syncleus Ferma[6] was used as an extension to Tinkerpop. 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 3.1.

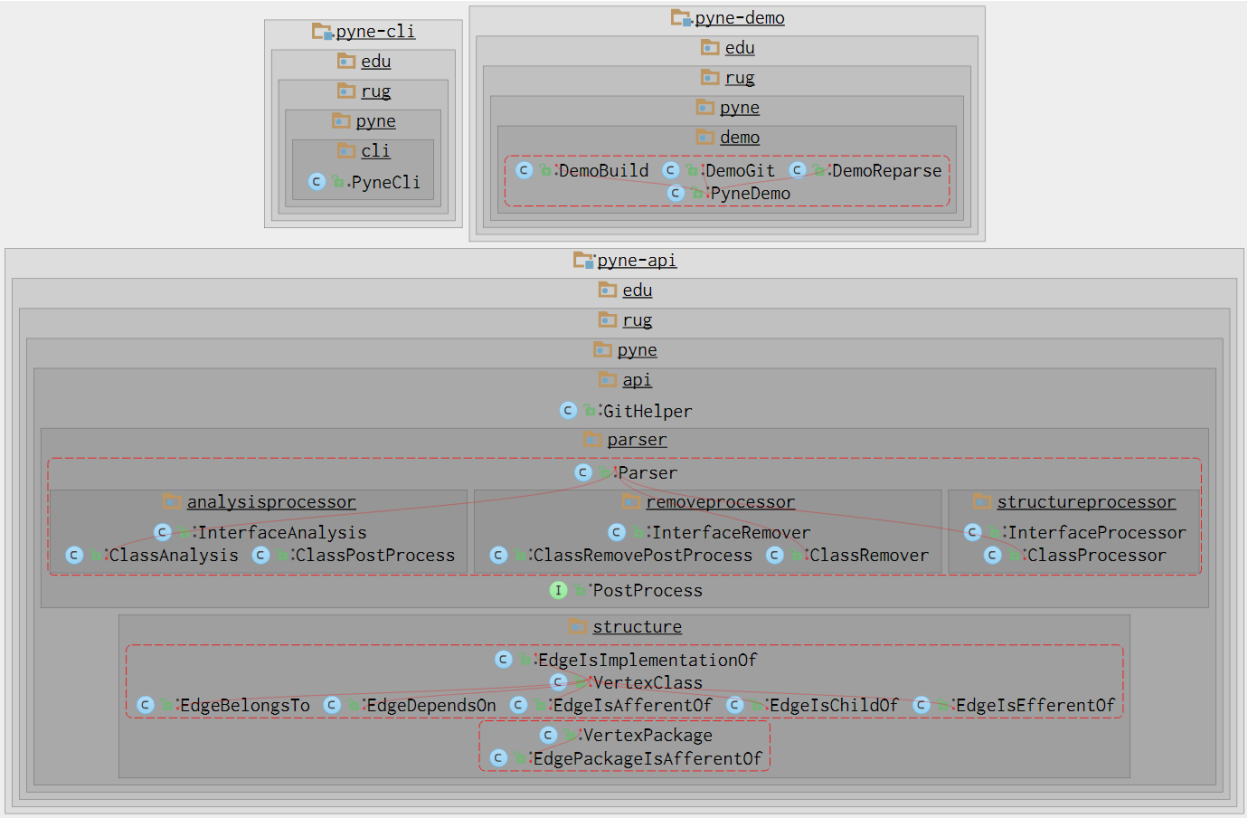

Figure 3.1: The full graph of the relations between the classes of Pyne, created by Structure101

explain figure 3.2

The purpose of these 3 packages will be briefly explained here. The pyne-api package will be explained in more detail, since this is the main component of the system.

[4] https://www.eclipse.org/jgit/
[5] https://commons.apache.org/proper/commons-cli/
[6] https://github.com/Syncleus/Ferma

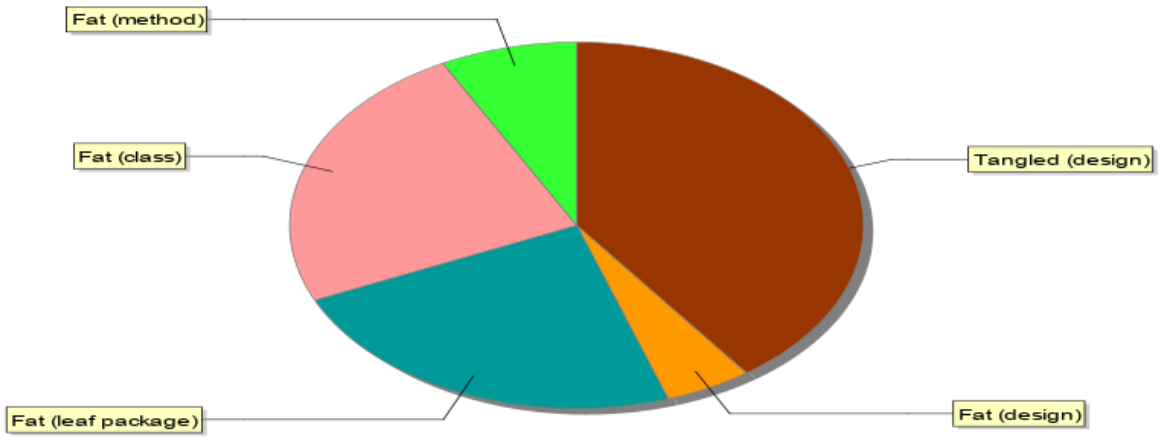

Figure 3.2: The XS diagram of the structure of Pyne, created by Structure101

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 and it is not present within the project, 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 pyne-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.

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

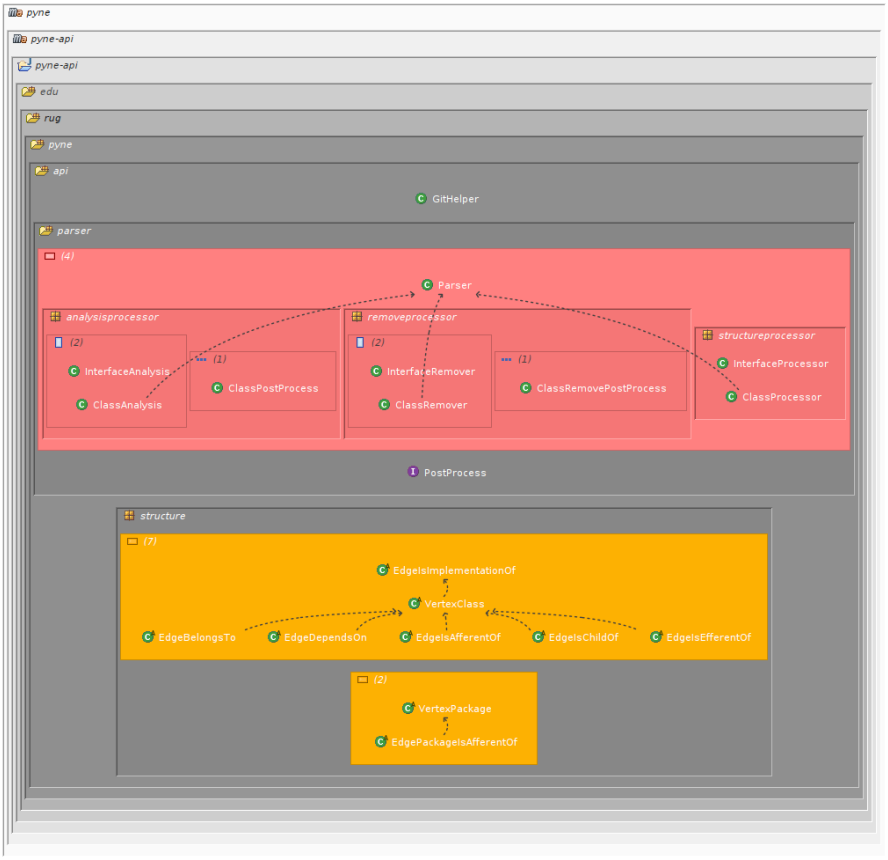

Figure 3.3: The graph of the relations between the classes of the pyne-api package created by Structure101

relations: BelongsTo, DependsOn, AfferentOf, ChildOf, EfferentOf, ImplementationOf and PackageisAfferentOF. Special vertexes 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 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 suppose 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 build from scratch[1].
- **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. Used to log actions in the program.
- **Ferma**
  An extension to Apache Tinkerpop. Used to help building the graph.

Most of these dependencies were noted in the report [1], 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 really relevant to the workings of Pyne itself.

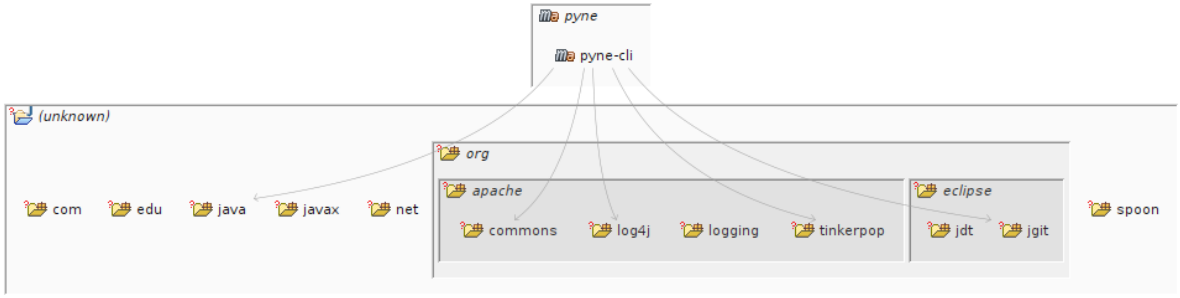

Figure 3.4: The graph of the external dependencies of pyne-cli, created by Structure101

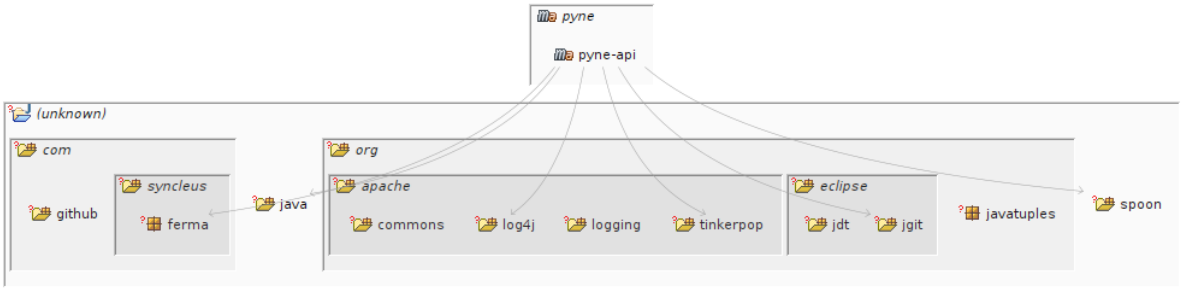

Figure 3.5: The graph of the external dependencies of pyne-api, created by Structure101

Chapter 4

Pyne vs Structure101

To compare Pyne to Structure101, we will test them both 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 Pyne and structure101. Note that in the commands we show, 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. This was however not indicated clearly, so it took some time to figure out.

However, by being able to use a maven pom file, Structure101 does have the advantage of being able to find dependencies to external packages instead of only internal packages, like Pyne does.

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.

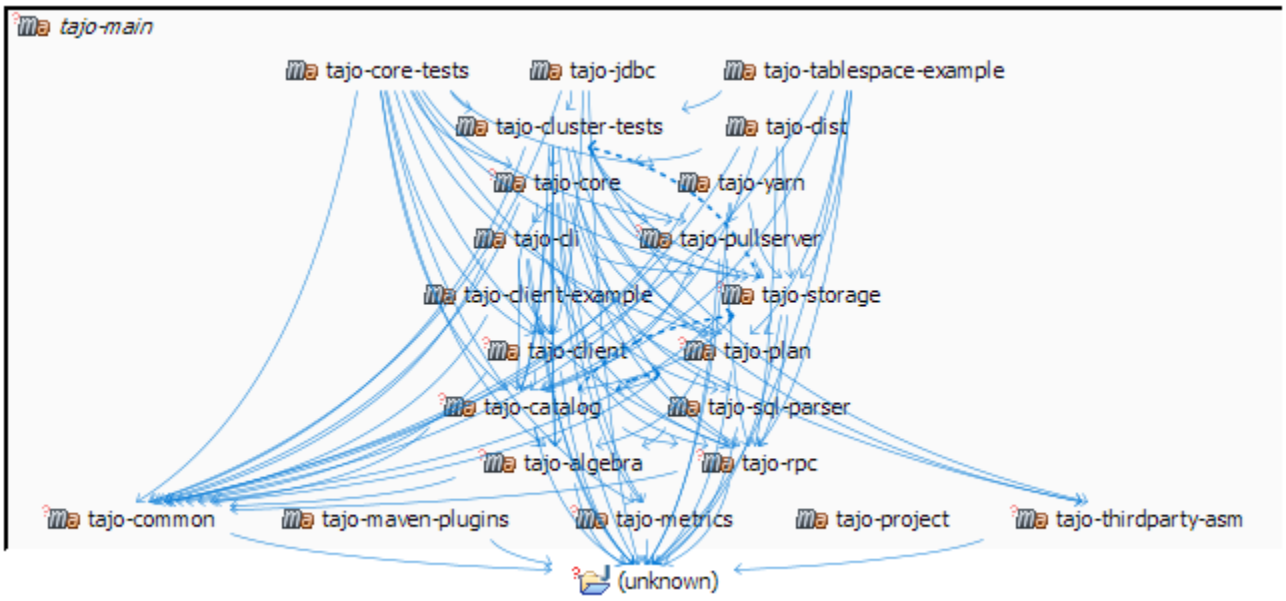

Figure 4.1: The graph of Tajo, created by Structure101

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 is however the dependency of tajo-storage (more specifically the internal tajo-storage-kafka and tajo-storage-pgsql packages) on tajo-cluster-tests. This dependency was however caused by the fact that tajo-cluster-tests was added in as a dependency in the pom.xml file, while non of the

classes actually needed it. In other words, this dependency could've been removed from the pom.xml 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 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 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, depend 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[4] definitions and some helper functions. Since this is basically a util package, it does not depend on other components. Not surprisingly, 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. Tajo-client has a few minor incoming and outgoing dependencies, but the most notable one is that of tajo-core with 460 references. It goes without saying that 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 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 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 handeling 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 just provides a few functions for tajo-core and the two test classes to apply a metric. Therefore only those classes depend on it and 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 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 presumable an extension of tajo in order to use it with yarn[5]. 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.

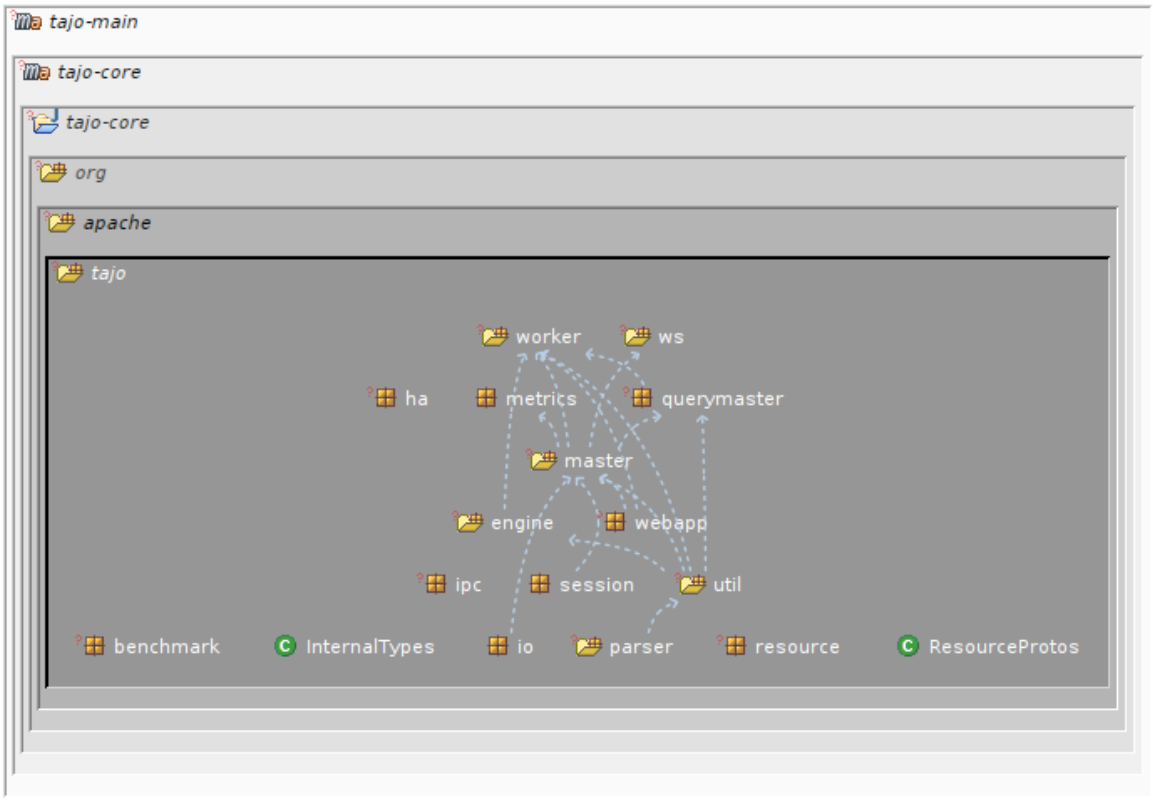

Figure 4.2: The graph of Tajo-core, created by Structure101

4.2.2 External dependencies

Here we will list a few of the main dependencies of tajo. This list will mainly include the external dependencies 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 that to google protocol buffers[4]. 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 amazonaws[6]. 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 tajo-core is codahale[7] and gmetric4j[8]. 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 tajo-core is minidev.json[9], which provides functionality of reading and parsing json-documents. The dependency jayway.jsonpath[10] is used in combination with minidev.json in a few minor cases to read the json documents.

Another remarkable dependecy is the maxmind.geoip library[11]. 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 your personalised location data for ad revenue.

two important dependencies for tajo-cli are the JLine library[12] and jansi[13], which provides functionality to handle and format console input.

antlr[14] is also used by both 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 tajo-core, it seems to be only used for SQL parsing.

A very important dependency of tajo-core is hadoop[15]. Not to mention that it is even required to have hadoop installed in order to use apache tajo. Hadoop provides a software framework for distributed storage and processing of big data.

Another important dependency used by tajo-core and tajo-rpc is glassfish.jersey[16]: An open source framework for developing RESTful Web Services in Java.

Snappy[17] is a dependency used by tajo-common for decompression/compression with the aim of highspeed 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.

Tajo-core uses mortbay jetty[18] as a java web server. More particularly Tajo-core uses this framework for their machine to machine communications.

Tajo-core also uses reflections[19]. Reflections scans your classpath, indexes the metadata, 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[20] 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.

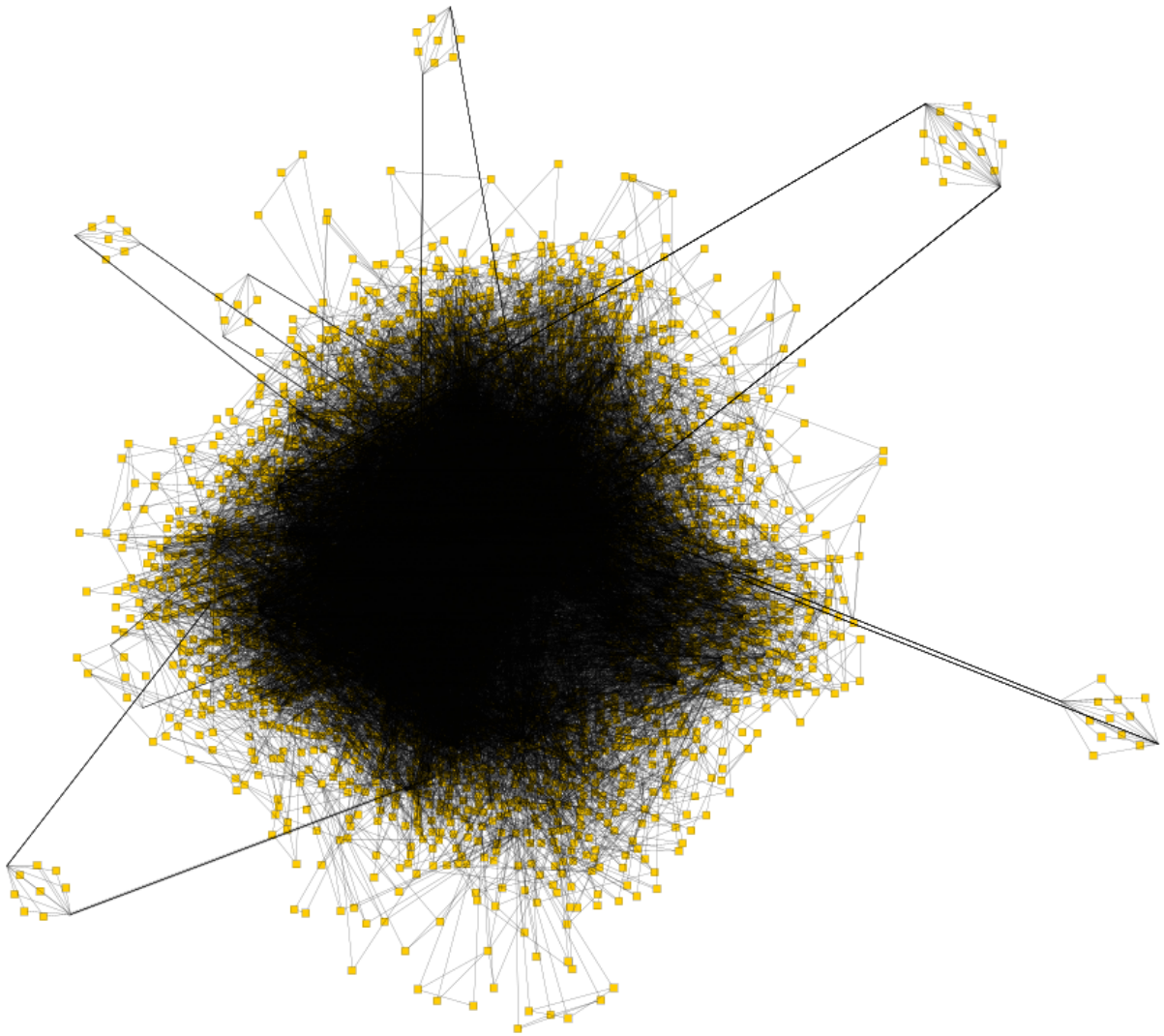

Figure 4.3: The automatically rearranged graph of Tajo, created by Pyne and rearranged by yEd

The nice thing about Pyne though, is that it can generate this 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 in the next section.

[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 source	NO	YES
Shows external dependencies	NO	YES
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

Table 4.1: Comparing the features of Pyne and Structure101

4.5 Comparing the Results of Pyne to Structure101

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, as mentioned in section 4.1, Structure101 does have the option to export to `.xml`, which we didn't investigate much further then, since it was incompatible with `.graphml`. But after exploring the exported `.xml` from Structure101, we came to the conclusion that we could still use this as our machine parsable format that we needed.

So we wrote a small java program[4] which extracts the found dependencies from both of these XML files[5], and checks if there any dependencies that were found by one tool, but not by the other.

note however that this is program is still quite basic, and will be expanded later (also to be more user friendly)

Its output first contains n lines of the following format:

```
[DEPENDENCY_NAME] ([in/ex]ternal) was missed by [TOOL_NAME]
```

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

[4] Available in our GitHub repository, under `dependency_checker/`

[5] `.graphml` is a version of XML standardised for storing graphs, meaning it is technically still valid XML and can therefore be parsed by any XML parser

Where `[DEPENDENCY_NAME]` is the name of the missed dependency, `[in/ex]ternal` indicates whether this is an internal or external dependency, `[TOOL_NAME]` is the name of the tool that missed this dependency, and n is the total amount of dependencies that were not found by all tools.

Finally at the end, the program outputs a summary for each tool, in the form:

```
[TOOL_NAME] missed X dependencies that the other tool did find
```

Where [TOOL_NAME] is the name of the tool, and `X` is the amount of dependencies it missed compared to the other tool.

Currently, before making any modifications to Pyne, we get the following statistics:

```
PYNE missed 199 dependencies that the other tool did find
STRUCTURE101 missed 1 dependencies that the other tool did find
```

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[6]. The reason why it didn't get labelled by its full name (`org.apache.parquet.schema.PrimitiveType.PrimitiveTypeNameConverter`) is unknown, but it might indicate a bug in Pyne.

further research needed

But of course more importantly, we see that Pyne missed quite a lot of dependencies that Structure101 was able to find.

Looking a little further into this, we stored the full output of Strcuture101, Pyne, and our comparison program, and looked through all lines of missed dependencies. Counting the amount of internal and external dependencies missed by Pyne, we found that 16 out of the 199 were internal and the remaining 183 dependencies were external. Looking through the `.graphml` file from Pyne, there are some external packages in there, and there seems to be no pattern in which dependencies are missing at first glance. So we will have to perform a more in-depth analysis on why Pyne didn't find the other external dependencies. This could be possibly be achieved using Structure101, or by investigating Tajo's and Pyne's source code.

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

Appendix A

Troubleshooting

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

1. **Bug in graph creator**:
   When I open a graph in a graph editor like draw.io or yEd graph editor, only one square appears. But the file is 2.25mb. Turns 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 itself 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. I sincerely hope that an integer would be enough to count the amount of lines of code anyway, so we'll change that.

2. **Default cli values**:
   The default options for 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. We shall therefore change the default values.

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 version 1.8 is needed. This leads to confusing errors if you try to compile Tajo for yourself after upgrading to java 11 to compile Pyne.

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).

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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