Analyzing dependencies between Java units of code

Team Members (group 24)

- Daniel Kerbel Daniel.Kerbel@mail.huji.ac.il CSE: **danielkerbel**
- Dan Shumayev Dan.Shumayev@mail.huji.ac.il CSE: **dans**
- Shaimaa Maranah Shaimaa.Maranah@mail.huji.ac.il CSE: **shaimaa.marana**

Problem Description

In this project we wanted to analyze the way that various Java hierarchies - units of code like methods, classes and packages, depend on each other, and to be able to visualize these dependencies.

Namely, we were interested in two approaches:

1. Analysis of the function call graph: When function A calls function B, this implies a direct dependency between these two functions, as well as the classes and packages that contain them.
2. Analysis of diffs in the Git commit log: We believe that units of Java code that are changed together (in the same commit, or several commits within a pull request) might also be dependent in some way.

We tried each of these approaches in order to generate a graph of Java hierarchies(methods/classes/packages) – each modelling a different kind of dependency (function calls vs being committed together), and then to visualize them in an interactive manner, with the hope that this visualization would be useful in learning the structure of the program – learning what are the important units of code, or how it can be refactored.

Data

The models that we'll implement aren't inherently limited to a specific dataset and are designed to work with any Java repository/program, with some caveats:

1. The analysis of call graphs requires .jar files that that were compiled for Java 8 (target compatibility 1.8) or below, and we only support executables and not libraries (must specify a main class that contains the main method)
2. Analyzing pull requests works for GitHub repositories only.

Moreover, since both of these tasks are computationally demanding, they only work with small-medium projects. For the sake of our evaluation, we have chosen the following repository:

https://github.com/skylot/jadx - Jadx is a Java decompiler that also supports Android APK files

It has 1660 commits, 356 pull requests. Its call graph has 5273 vertices and 24040 edges when we only consider function calls that involve 'jadx' package (and not other library calls).

Solution

We'll present each of the two approaches which are mostly orthogonal, and the shared graph/visualization step.

Definition – "Java Hierarchy"

A java hierarchy is a certain level of organization for Java code, we consider 3 different hierarchies:

1. Package is the highest hierarchy, a Java file can only contain one package, several files may share the same package. Packages also have a hierarchy, e.g, `com.foo.pakA`, but for our analysis we ignore this hierarchy, and consider each package identifier as its own.
2. Type declaration: class, interface or an enum.
   A package has one or more type declarations, and type declarations might also be nested within type declarations. We make no distinction between classes/interfaces/enums nor do we care about their inner hierarchy – each type declaration is considered on its own.
3. Function: method, static method or constructor.
   A type declaration may have one or more functions.

We can control the level of granularity when using our algorithms: If we created a graph between functions, we can also treat it as a graph between type declarations or packages.

Generating a call graph

In this method we used the Soot library in order to generate a call graph for a Java executable. Soot is a library mostly intended for Java optimization and can generate a lot of information – in our case we're only interested in method calls. It should be noted that due to Java's use of dynamic dispatch (runtime polymorphism) or reflection, it is computationally difficult to determine all kinds of method calls via static analysis. Soot uses an algorithm based on "Class Hierarchy Analysis", which is essentially a heuristic that uses the class hierarchy in order to shrink the number of candidates in a polymorphic method call.

Using this method yields us a directed, unweighted graph: If method A calls method B, this implies that A depends on B but the inverse is not necessarily true. We can also constrict the graph into class or package dependencies by combining nodes. In this case, we do treat the graph as weighted by summing the number of combined edges – this way the weight indicates the strength of the dependency between the hierarchies.

Processing git diffs in commits/pull requests:

For our first approach, we were interested in finding some kind of correlation between Java hierarchies and the commits/pull requests in which they were modified. This seemed like a good fit for association rule mining, and involved several steps:

Feature Extraction – from diffs to item baskets

Git by itself has no understanding of source code: when it generates a diff (a textual representation of the changes between two versions of a repository), the diff is based on raw text differences between lines. Therefore, we had to parse the source code which was modified in a diff in order to obtain a more high-level understanding of what changed: to be able to match a byte position to the identifiers of the Java method, class and package that contains it.

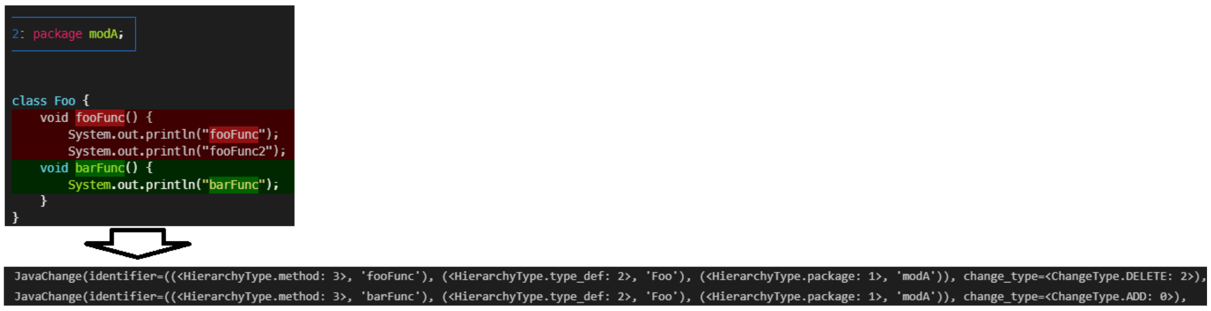
There were several options we considered, with different impacts:

- Using regexes to scan class/method declarations. This is the fastest solution, however regular expressions cannot deal with recursive languages like Java, namely, they cannot match the beginning and end of brackets (which might be nested) – required in order to determine the boundaries of each declaration.

- Using a pre-existing Java parser would've been ideal, but would be too slow for our purposes, and difficult to integrate in Python. We are only interested in class and method declarations and their scopes, whereas a parser obtains the entire parse tree, including method body and other syntactic elements we don't care about. Since we have to parse thousands if not tens of thousands of files (each commit may have multiple files) – speed was especially important.
- We eventually decided to implement our own parser. The parser consists two stages:
    - Lexing: Converting the text within the diff into Java tokens: identifiers, keywords, operators, literals and comments. This was done via Regex. Each token also includes the token's start and end position.
    - Parsing: Converting the tokens into a parse tree (the "JavaHierarchy" object). The parser is based on the concept of recursive descent, however in contrast to a full parser which expects a rigid grammar, our parser involved a lot of skipping in order to ignore elements that we don't care about – the contents of a method's body, fields, modifiers, generics, types, fields, etc.

After parsing, we have a parse tree (represented via `JavaHierarchy` object) where each node also includes the byte offsets of the node's start(declaration) and ending (semicolon or closing bracket). Traversing this tree in DFS preorder (topological sort) yields the nodes in the order they appear in the text. Combined with the start/end offsets – this allows us to match the byte offsets of a git hunk (a set of consecutive lines that were added/deleted) into semantic, Java changes.

Here are some basic examples that summarize the process:

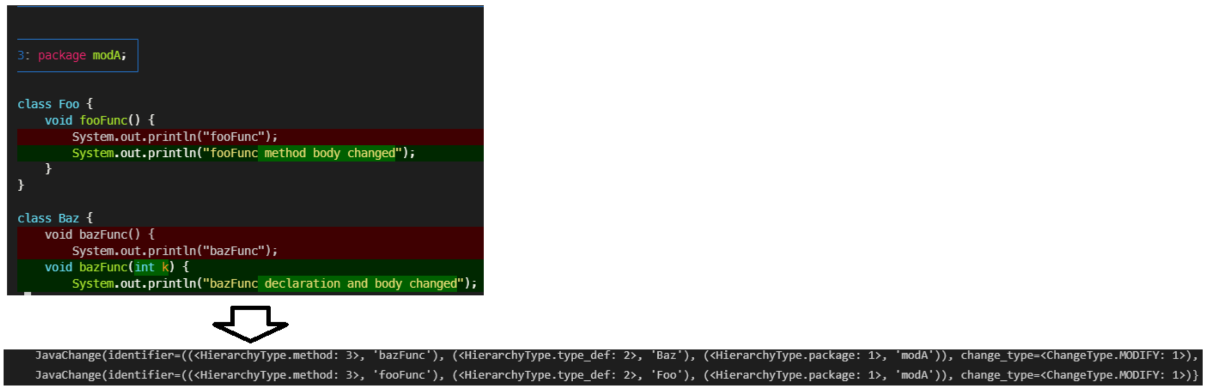


```
2: package modA;


class Foo {
    void fooFunc() {
        System.out.println("fooFunc");
        System.out.println("fooFunc2");
    void barFunc() {
        System.out.println("barFunc");
    }
}
```

```
JavaChange(identifier=((<HierarchyType.method: 3>, 'fooFunc'), (<HierarchyType.type_def: 2>, 'Foo'), (<HierarchyType.package: 1>, 'modA')), change_type=<ChangeType.DELETE: 2>),
JavaChange(identifier=((<HierarchyType.method: 3>, 'barFunc'), (<HierarchyType.type_def: 2>, 'Foo'), (<HierarchyType.package: 1>, 'modA')), change_type=<ChangeType.ADD: 0>),
```

An identifier is essentially a list of Java Hierarchy names, from most specific to most general.



```
3: package modA;

class Foo {
    void fooFunc() {
        System.out.println("fooFunc");
        System.out.println("fooFunc method body changed");
    }
}

class Baz {
    void bazFunc() {
        System.out.println("bazFunc");
    void bazFunc(int k) {
        System.out.println("bazFunc declaration and body changed");
```

```
    JavaChange(identifier=((<HierarchyType.method: 3>, 'bazFunc'), (<HierarchyType.type_def: 2>, 'Baz'), (<HierarchyType.package: 1>, 'modA')), change_type=<ChangeType.MODIFY: 1>),
    JavaChange(identifier=((<HierarchyType.method: 3>, 'fooFunc'), (<HierarchyType.type_def: 2>, 'Foo'), (<HierarchyType.package: 1>, 'modA')), change_type=<ChangeType.MODIFY: 1>)}
```

We say that both ‘Foo.fooFunc’ and ‘Baz.bazFunc’ were changed, since their contents were both added and deleted within the same commit. (Note that in our analysis we didn’t actually care about the type of change and only used the identifier, the change type was mostly used for debugging)

Association Rule Mining & Graph Construction

After converting each diff to a list of “JavaChange”, we began association rule mining:

- The baskets are, depending on the user’s choice - commits or pull requests
- The items are identifiers of Java hierarchies (method/class/package names) which were changed in a PR/commit. The level of granularity also depends on the user’s choice.

The intuition here is that Java hierarchies which are related will often be changed together (in the same commit or PR): an association rule X->Y indicates that often when X changes, so does Y, in other words - Y depends on X. Therefore, we can model this kind of dependency via a graph, creating an edge (Y, X) whose weight is the confidence of the rule. The direction of the graph matters because association rules aren’t necessarily symmetric.

A useful observation is that we don’t need to use the association rule algorithms to their full extent. We only care about rules between singletons, as we are drawing edges. Rules of the form {X} -> {Y} can be generated from itemsets of lengths 1-2, thus when using the Apriori algorithm, this cuts down the memory complexity to $O(f^2)$ where f is the number of different items (Java hierarchies).

Note that we still need to supply a threshold for a minimal support and confidence. This depends on the provided repository, and we didn’t find a way to automatically determine these numbers. So, there’s a process of trial and error in order to generate a graph that is large enough to be useful, but not too large to handle from a computational aspect.

Processing and visualizing the graphs

In both of these approaches we generate graphs representing some sort of dependency or relationship between Java hierarchies. We apply two algorithms in order to aid in the interactive visualization of these graphs:

Page rank:

In the call-graph model, the page rank essentially represents the likelihood of a given function appearing in a random function’s call-stack. Generally, a call graph of a single method looks like a tree (assuming no mutual recursion) and while the call graph of all methods might be more complicated, the tree-like trend implies that with a large damping factor – the returned results will often be leaf nodes: often these will be utility methods and common data structures, this may or may not be useful to the user based on the context, therefore we didn’t settle on a specific damping factor – a smaller factor(more random teleports) might yield more useful results in some cases.

In the association rule model, a node with a large page rank represents an item that appears as the antecedent of many association rules, whether directly or indirectly. Overall, page rank alone isn’t very useful, and it’s mostly used in the visualization to control the size of the node, as well as for ordering when adding nodes.

Community detection

We use the “Leiden” algorithm for community detection, which is based on optimizing graph modularity (a heuristic approach, since it’s NP hard to maximize modularity). It’s an evolution of the popular “Louvain” algorithm, known to be robust and efficient for large networks too, and works with directed and weighted graphs – in contrast to Girvan Newman

A clustering with high modularity is one where nodes within each community are densely connected, but connections between different communities are sparse.

The parameters of this algorithm are resolution (0-1), and number of iterations. Both are configurable by the user. Large resolutions yield more small communities whereas low resolutions yield fewer bigger communities. Increasing the number of iterations can only improve the clustering's modularity – at the cost of longer running time.

Evaluation & Visualization

Criteria & Impediments

It is difficult to quantify the performance of our models in an objective manner, for several reasons:

- There is no objective metric we're trying to maximize; the output of the model is ultimately a graph that's meant to be a representation of the indirect dependance between Java hierarchies – whether inferred from call graph or by associated commits, it's hard to judge whether they truly convey the dependence of those hierarchies.
- The output depends on the java program/repository being analyzed: The way the code is organized into hierarchies and the pattern by which changes to the code are committed. Therefore, we cannot reliably measure the quality of our model among different repositories, nor is there any specific repository that is an ideal candidate for our measurements.

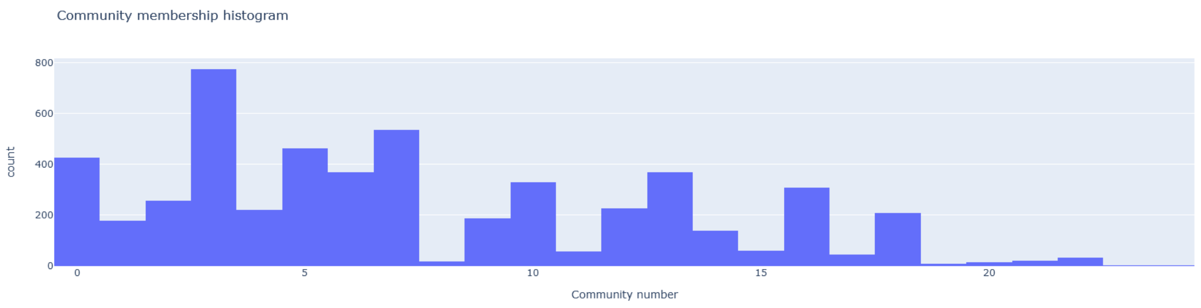
With that being said, there are some metrics that can help us choose parameters for the association rules model – such as support and confidence, and for both of these models, we can use modularity to gauge the quality of the clustering of the final graph. The modularity of a clustering is the fraction of edges that are inside clusters minus the expected fraction of such edges for a random edge assignment. It is a value between -0.5 and 1.

Modularity can also be seen as a metric of the design of the code-base itself: a dependency graph with dense clusters that are sparsely separated, hints that we have several modules that are well isolated/loosely coupled – a property that is often desirable. Conversely, low modularity might indicate tight coupling and "god objects" which are anti-patterns.

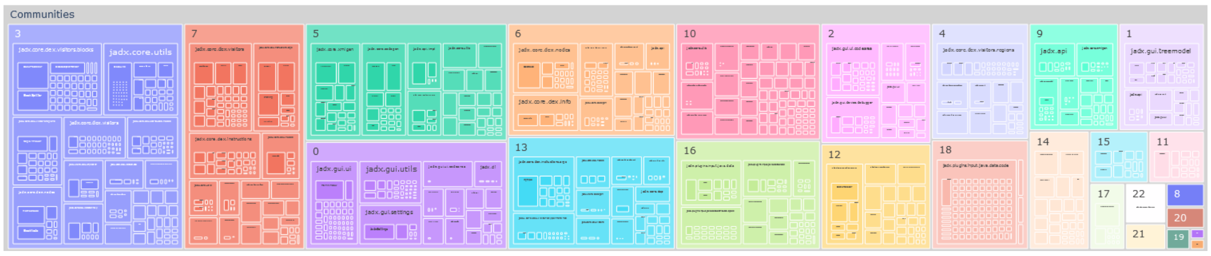
Setup & Evaluation

Call Graph model

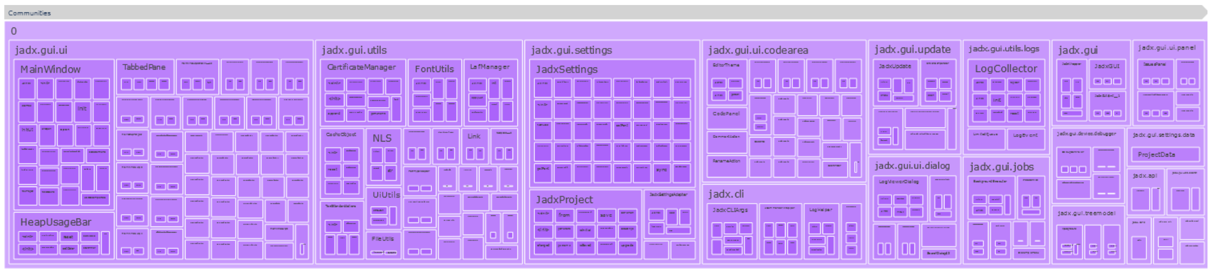
First, we compiled Jadx and created a call graph using our web frontend. The entry point was the GUI program (though there also exists a CLI version). We filtered to files and edges that contain the keywords "jadx" as we aren't interested about external library calls.
Then, after importing the graph(using the same web frontend) and using 500 iterations for the community detection algorithm (taking about 20-30 seconds to compute) and a maximal resolution 1.0, we obtain a modularity value of 0.645, indicating that the clustering is better than random assignment. The visualization tool also presents some useful plots about the community distribution:



And an interactive tree-map visualization, which can be thought as a more detailed form of histogram, using communities as the first level of hierarchization, followed by the actual Java hierarchy (packages, classes and methods):

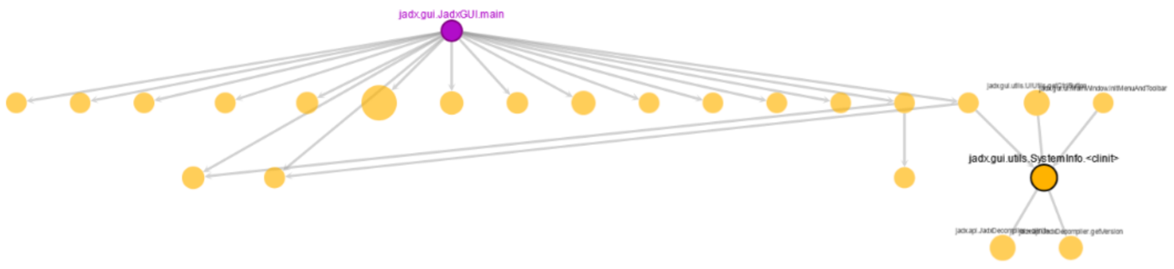


This is a very useful way to learn about the structure of the program and to get a quick overview of the different hierarchies in the program, based on the frequency they appear in function calls. We can click on any level to get a more in-depth view:

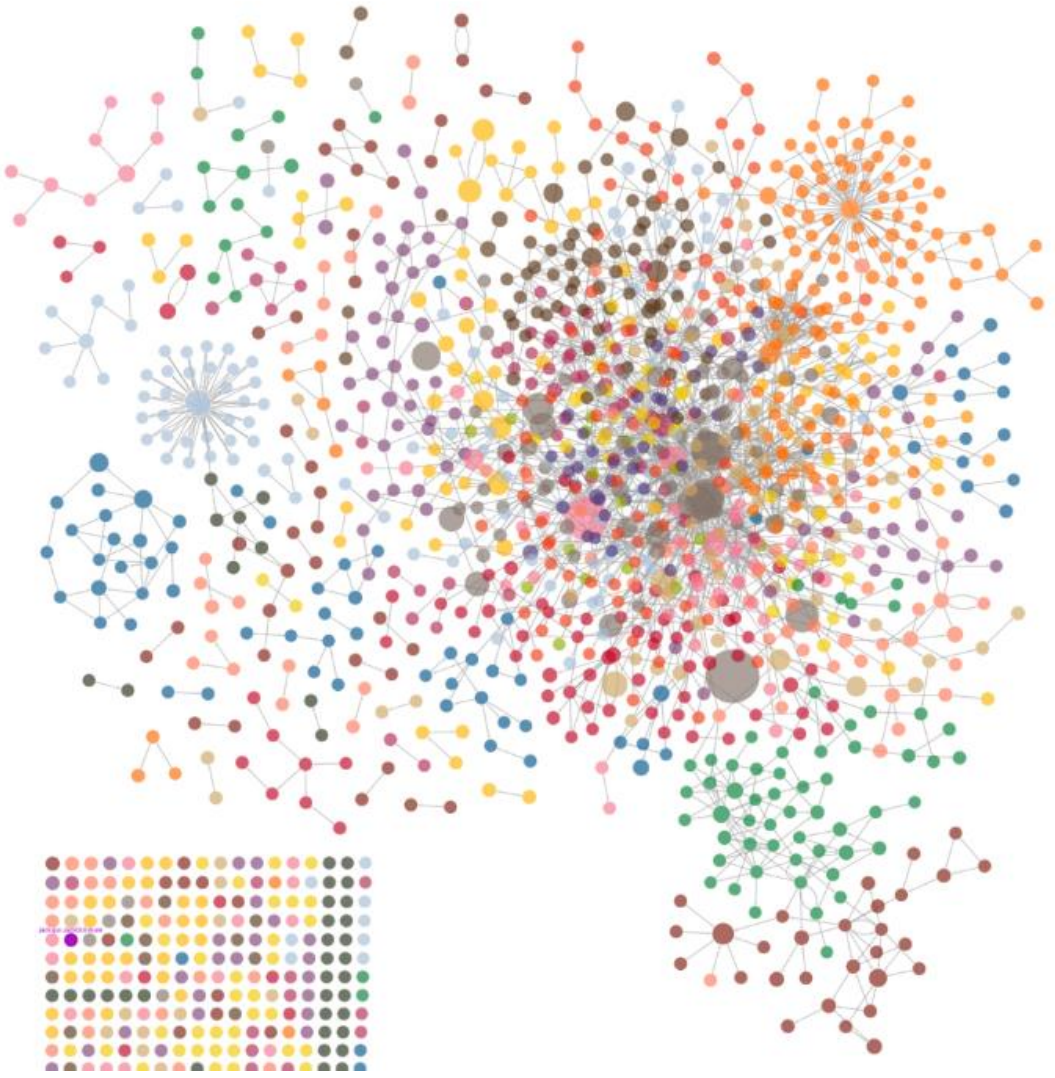


Note how most of the GUI related packages, as well as the CLI entry point, were placed into the same community. The graph's modularity is 0.645 – indicating the clustering is quite better than a random assignment.

The graph view begins with the first node in the graph – which happens to be the entry point. You can a node to expand it, and there are various graph layout options. For example, using 'dagre' illustrates the tree-like behavior in a more compact way:


jadx.gui.JadxGUI.main
jadx.gui.utils.UiUtils.openChildren
jadx.gui.ui.MainWindow.initMenuAndToolbar
jadx.gui.utils.SystemInfo.<clinit>
jadx.api.JadxDecompiler
jadx.api.JadxDecompiler.getVersion

You can also focus on a specific node using the control panel, or add many nodes. Using the 'fcose' layout – a force directed physical layout, makes the community division more clear. Using 2k nodes(and only edges between these 2k, hence some nodes that don't have neighbors in this viz):



And you can also lower the granularity, from methods to classes or packages, which might help in readability.

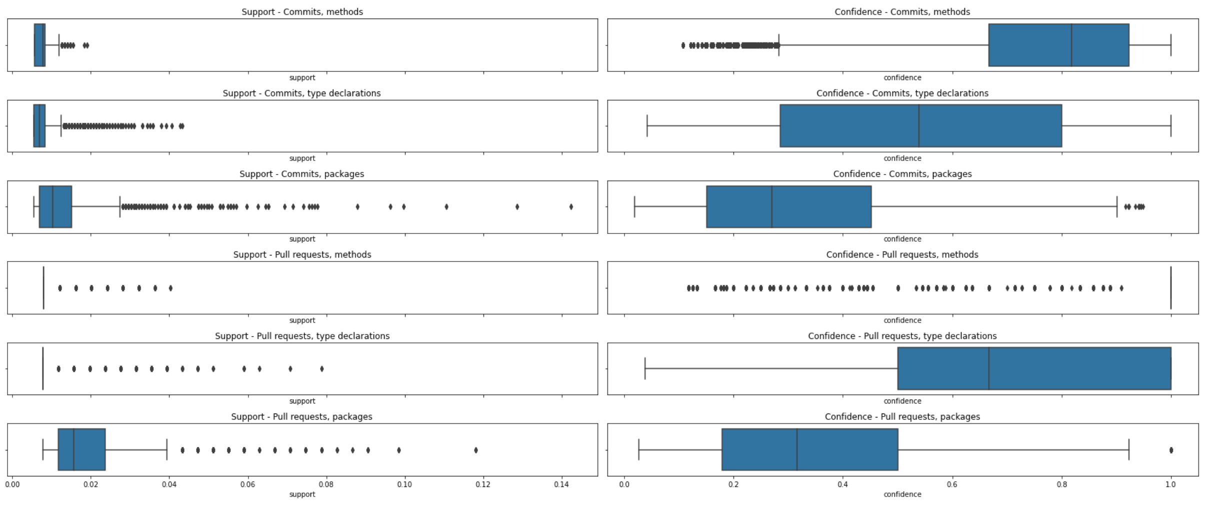
Association Rules

First, we extracted some preliminary statistics about the number of items and baskets in each model:

		#baskets	**#items**
basket_type	**item_type**		
commit	method	1423	13359
	type_def	1452	2888
	package	1455	192
pr	method	248	4343
	type_def	254	1072
	package	254	106

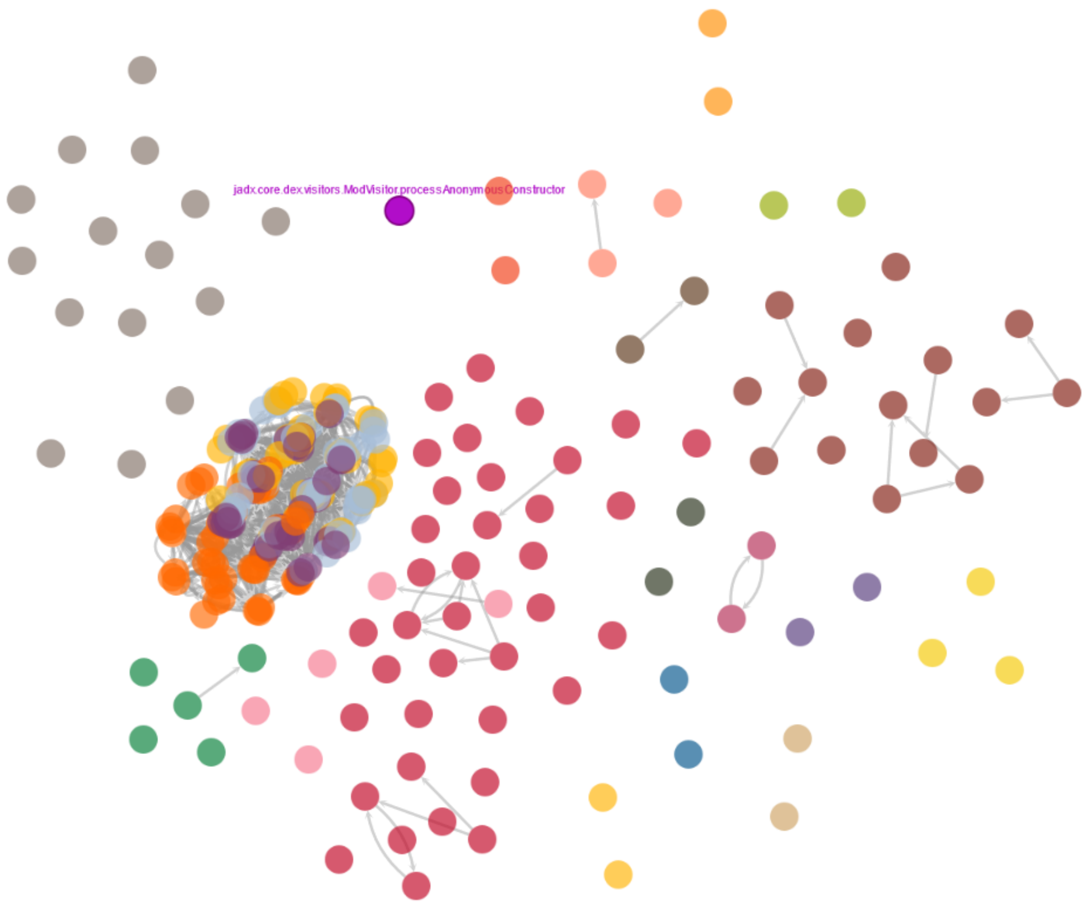
We can see that many changes were not done via PRs but as lone commits, as the set of different Java hierarchies changed in commits is much higher than those via PRs. This is probably explained by the author of the git repository making direct commits(without PRs), but also because a PR diff skips many intermediate/temporary commit which may have introduced new identifiers/hierarchies that were later dropped.

We then computed frequent itemsets and association rules. The minimal support had to be quite low(around 0.005) to get a significant amount of results, and a minimal confidence of 0 was computationally viable(filtering by weights is done during visualization) . The following plots summarize the distribution of the support and confidence values of the association rules:



The small number of baskets in the PR model, coupled with low support values – indicates their results are not well representative. Using commit with method items gives an acceptable support(around 1%, considering the large number of items this seems reasonable) and high confidence, though we had expected to see more rules with higher supports – we believe the reason for this discrepancy is the fact our model does not handle refactors (Java Identifiers that are renamed or moved to different hierarchies, etc..) – this also explains why the number of hierarchies is much higher than the call-graph, which only contains the hierarchies present at the latest version, and not over the course of the repository's lifetime.

We attain a modularity score of 0.382 when using commits and methods, which is worse than the callgraph model, but better than other item & bucket combinations. Here's the resulting graph after filtering to edges with 95%+ confidence


jadx.core.dex.visitors.ModVisitor.processAnonymousConstructor

Conclusion

In this project we dealt with fairly complicated feature extraction pipelines in order to generate graph data, utilizing association mining techniques and community detection algorithms, and created an interactive visualization for these models. Ultimately, the association rule model did not perform as well as we expected, but we believe there are ways to bridge this in future work:

- Find a better way of determining baskets, instead of using commits or PRs, perhaps try grouping commits/PRs – e.g, by author or by textual elements that appear in commits(some form of topic detection or maybe NER?)
- Find a way to deal with refactoring in a git repo's lifetime, e.g, match slightly changed identifiers or hierarchies based on common keywords or even code itself
- Improve the visualization tool: better performance, better interactivity, more filtering options, consider other metrics for node importance besides page-rank.