Arcan 2 CLI User Manual

(Arcan version 2.8.5-RELEASE 2023-01-27)

Introduction

Arcan 2 is a full rewriting of Arcan 1 that includes several new features that complement the analysis of architectural smells in software systems.

What's new?

The new features include:

- Multiple programming language support: Java, C, C++, C#, and Python (planned)
- Multiple analysis modes:
  - Analyse one snapshot (e.g. commit, version, or release) at a time and save the status of the analysis to be resumed later (useful for users)
  - Analyse multiple versions in one call (useful for researchers)
  - Analyse multiple versions in one call from a list of commits (useful for both researchers and users)
- Track of architectural smells and source files over time for a precise evolution analysis
- Automatic calculation of thresholds based on frequency analysis
- Fine-grained filtering of the source code files and directories to be analysed with versioning support.
- Multi-command (i.e. git-like) command line arguments design
- Extensive help and description of each parameter (user-friendly)
- Custom fine-tuning of smell detection thresholds
- And many more.

Arcan commands

Arcan 2 features a git-like command line interface, namely multiple commands are available to the user to perform different tasks:

- `analyse`: Analyse a project to detect and track architecture smells and components (packages and classes). This is the main command you are going to use. `analyse` has the following subcommands:
- `track`: Track components and smells using GraphML files as input. This command is used if one wants to separate smell detection and tracking (usually for performance reasons). The `analyse` command must be executed first.

- `generate`: Generate sample configuration files to use a basis for your analysis. `generate` has the following subcommands:
  - `filter`: Generate a file that describes what source code directories to include/exclude in the analysis.
  - `includeMakefile`, `include`: These are two commands to generate files that let Arcan know how include directives should be resolved when analysing C/C++ projects.
- `dry-run`: Allows to check exactly what directories are going to be analysed by the analyse command.
- `options`: Prints to the console the full list of configuration options that the user can tweak when running the analyse command.
- `intro-order`: convenience command not discussed in this document.

All commands have a `-h` option that prints to screen a help message concerning their parameters.

Step-by-step analysis examples

In this section, we describe how to perform a full analysis of a real project. Use this example to guide you on how to perform the analysis of a project. Remember to read carefully the help message of the command `analyse` to fully understand what you are doing. Note that, if you are not running Arcan in a Docker container, you may also need to install GNU `libgomp` on your machine, which is most likely available via the package manager of your Linux distribution (e.g. `sudo apt install libgomp1` on Ubuntu).

Let's start by cloning ANTLR 4, our example project:

```
$ cd /tmp # Optional command
$ git clone https://github.com/antlr/antlr4.git
```

To simplify your command line, we recommend that you define the following alias:

```
$ alias arcan="path/to/arcan.sh" # On Linux

> doskey arcan="path/to/arcan.bat" # On Windows
```

now you can invoke Arcan simply by typing `arcan` in your terminal. Moreover, you can also add this at the end of your `~/.bashrc` file so that it is always available to you, even after restarting.

Note that the `-add-opens` option is necessary due to a library used by Arcan that necessitates access to `AtomicLong` class via reflection. This is a temporary workaround until the library officially fixes this.

Simple analysis

Now, let's analyse it as-is without filtering any source code directory:

```
$ arcan analyse -i antlr4 -p antlr -o /tmp --all -l JAVA
```

this will analyse antlr and save the output CSV files to `/tmp/arcanOutput/antlr`.

Let's break down each argument:

- `-i` tells Arcan what is the input directory of the project.
- `-p` tells Arcan what is the name of the project. This name is going to be used to create the output directory where output files are going to be saved. Output files will also have a column with the name of the project.
- `-o` this is the directory where to create the `arcanOutput` directory.
- `--all` tells Arcan to detect all smells currently supported. **Note** that if you do not provide this, no smells will be detected.
- `-l JAVA` tells Arcan that it has to look for Java files.

The output directory will contain the following files

```
$ ls arcanOutput/antlr
component-metrics.csv  smell-affects.csv  smell-characteristics.csv
```

which are:

- `component-metrics.csv`: the list of components (again, classes and packages) and the metrics calculated for each one of them.
- `smell-characteristics.csv`: the list of smells and the characteristics (i.e. metrics for smells) calculated for each instance detected.
- `smell-affects.csv`: the list of what components are affected by which smells using the IDs from the `component-metrics.csv` and `smell-characteristics.csv` files, respectively.

If you are only interested in the smells detected, you should be good to go by looking only at the `smell-characteristics.csv` file.

In case you also desire Arcan to print the **dependency graph** as a GraphML file, you must run the following command:

```
$ arcan analyse -i antlr4 -p antlr -o /tmp --all -l JAVA
output.writeDependencyGraph=true
```

Fine-tuned analysis

When performing static analysis, it is sometimes desired to exclude certain files present in the repository. For instance, test files, example files, and external libraries might not be interesting to include in our analysis.

To do so, we can write what directories we want to analyse (or exclude from our analysis) in a *filters file*. Arcan allows you to generate this file so the only think you are left to do is writing the inclusion/exclusion patterns:

```
$ arcan generate filters /tmp/antlr-filters.yaml
```

This will write our filter as a YAML file to `/tmp/antlr-filters.yaml`. It contains something like this:

```
scanners:
- startDate: '0001-01-01'
  endDate: '9999-01-01'
  include: ['{**/src,src}/main/java/**', '{**/src,src}/java/**']
  exclude: ['{**/src,src}/test/**', '*testing/**', '*-test*/**', test/**,
'*examples/**']
```

This file tells arcan to analyse the paths that are included by the patterns in `include` and exclude (from the matched paths) the paths that match the patterns in `exclude`.

To make it clearer, let's take a look at ANTLR's directory structure:

```
$ tree -L 3 -d antlr4
antlr4
├── antlr4-maven-plugin
│   ├── resources
│   │   └── META-INF
│   └── src
│       ├── main
│       ├── site
│       └── test
├── build
├── doc
│   ├── faq
│   ├── images
│   └── resources
├── runtime
│   [...] # Contains source code directories
├── runtime-testsuite
│   ├── annotations
│   │   └── src
│   ├── processors
│   │   ├── resources
│   │   └── src
│   ├── resources
│   │   └── org
│   └── test
│       └── org
├── scripts
│   └── parse-extended-pictographic
├── tool
│   ├── playground
│   ├── resources
│   │   └── org
│   └── src
│       └── org
└── tool-testsuite
```

```
    └── test
        └── org
```

As we can see, there is a great variety of source code directories: source code, unit test, integration tests, and the maven plugin of ANTLR too! The actual ANTLR source code that we want to analyse is contained in `antlr4/tool/src`. Therefore we write the following in the `/tmp/antlr-filters.yaml`:

```
scanners:
- startDate: '0001-01-01'
  endDate: '9999-01-01'
  include: ['tool/src/**'] # This line includes all source code directories
we desire
  exclude: [] # No directories under tool/src that we want to exclude
```

Note that the patterns in `include` and `exclude` must conform to the glob pattern language. **All patterns must be specified relative to the root project directory** (`-i` parameter). Note that only the files **directly under the matched** directories are included in the analysis.

In case we want to check exactly what directories will be analysed if we used `antlr-filters.yaml`, then we run the following:

```
$ arcan dry-run -i antlr4 -f antlr-filters.yaml
[...]
Project source dir: antlr4 -> {
    /tmp/antlr4/tool/src/org/antlr/v4/codegen
    /tmp/antlr4/tool/src/org/antlr/v4/codegen/model/decl
    /tmp/antlr4/tool/src/org/antlr/v4/semantics
    /tmp/antlr4/tool/src/org/antlr
    /tmp/antlr4/tool/src/org/antlr/v4/parse
    /tmp/antlr4/tool/src/org/antlr/v4/codegen/model
    /tmp/antlr4/tool/src/org/antlr/v4/tool/ast
    /tmp/antlr4/tool/src/org/antlr/v4/codegen/model/chunk
    /tmp/antlr4/tool/src/org/antlr/v4/tool
    /tmp/antlr4/tool/src/org/antlr/v4/automata
    /tmp/antlr4/tool/src/org/antlr/v4/codegen/target
    /tmp/antlr4/tool/src/org/antlr/v4
    /tmp/antlr4/tool/src/org/antlr/v4/unicode
    /tmp/antlr4/tool/src/org
    /tmp/antlr4/tool/src/org/antlr/v4/analysis
    /tmp/antlr4/tool/src/org/antlr/v4/gui
    /tmp/antlr4/tool/src/org/antlr/v4/misc
}
```

To run the analysis on these directories only, we use:

```
$ arcan analyse -i antlr4 -p antlr -o /tmp --all -l JAVA --filtersFile
antlr-filters.yaml
```

Analysing multiple versions

Arcan also supports analysing multiple versions of a project by iterating over the commits in the Git repository of the project.

The analysis can be carried out in two different ways: by providing a list of commits you want to analyse, or by defining a range of dates from which Arcan will automatically select what commits to analyse.

List of commits

To analyse a list of commits, all you have to do is create a file where each line is a commit hash, with the oldest one on top.

```
$ cd antlr4
# Save the first 5 commits in antlr to a file
$ git rev-list --remotes | head -n 5 | tac > ../commits.txt
cd ..
# Run arcan
$ arcan analyse -i antlr4 -p antlr -o /tmp --all -l JAVA --versions
commits.txt
```

The file containing the commits is specified via the `--versions` argument.

Range of dates

Arcan can analyse the commits contained within a range of dates on a given branch:

```
$ arcan analyse -i antlr4 -p antlr -o /tmp --all -l JAVA -e --startDate
2018-01-01 --endDate 2018-03-01 --intervalDays 30 --branch master
```

With this command, Arcan will analyse all the commits within 2018-01-01 (YYYY-MM-DD) and 2018-03-01 that belong to the master branch and are at least 30 days apart.

Tracking smell instances and components

In some cases, it is interesting to also track individual smell instances over time. Tracking a smell instance means linking the same smell detected in two consecutive versions based on the artefacts it affects.

To run the tracking of both smells and artefacts, simply add the `-t` flag:

```
arcan analyse -i antlr4 -p antlr -o /tmp --all -l JAVA -e --startDate 2018-
01-01 --endDate 2018-03-01 --intervalDays 30 -t
```

After the command completes, the output directory will contain a `entity-tracking.csv` file, where each smell (or component) is assigned a unique id and linked to the smell ID in the specific version. The age of the smell (or component) is also printed out.

Alternatively, if the tracking proves to require too many resources, one can analyse and track in two separate phases:

```
# Analyse the project: option -t is omitted and the DependencyGraphs are
printed to file
arcan analyse -i antlr4 -p antlr -o /tmp --all -l JAVA -e --startDate 2018-
01-01 --endDate 2018-03-01 --intervalDays 30
output.writeDependencyGraph=true

# Track the smells and components using dependency graphs outputted by
previous command
arcan track -i /tmp/arcanOutput/antlr -o /tmp -d /tmp/antlr4 -b master
```

Analyse C/C++ projects

Analysing C/C++ projects is very similar to analysing Java projects. The only difference is that for C/C++ one needs to also provide a file that describes how include directives should be resolved. If the file is not provided, the include directives are resolved starting from the path of the file that uses them (i.e. default behaviour of most C/C++ compilers).

The file that needs to be declared is a YAML file structured as in the example below:

```
matchers:
- paths: [path/to/includes1, path/to/includes2]
  pattern: '**/path/to/implementors1/**' # Arcan will use the includes
paths above for files that match this pattern

- paths: [path/to/includes3, path/to/includes4]
  pattern: '**/path/to/implementors2/**' # For files matched by this
pattern, the paths above will be used instead.

[...] # Other matches, if needed
```

The file describes a series of matchers, each matcher has a pattern that's used to match files and the corresponding includes paths.

The example above can be generated using Arcan itself:

```
$ arcan generate include output-file-name.yaml
```

Alternatively, Arcan can perform a best-effort attempt to generate the file automatically:

```
$ arcan generate includeMakefile dir/to/makefile output-file-name.yaml
```

this command will try to look into the makefile of your project and extract the include resolution paths and save them into the given output file (that you can later use to analyse your project).

Dashboard integration

Arcan also provides the possibility to analyse a project via the command line tool and then visualize the results in a second moment through the dashboard.

To do so, it's necessary to ensure that Arcan stores the results in the same database file the dashboard uses. The dashboard, by default, stores the results in the home user directory under a hidden directory named `.arcan`. This directory can be located with the following commands:

```
# On Linux based systems run this
$ cd ~/.arcan

# On Windows run the following
$ cd %userprofile%/.arcan
```

Note that the directory will only be present if you ran the dashboard at least once.

Therefore, to save Arcan's results in this folder, we to run arcan with the `--jdbc-directory` option:

```
$ arcan analyse -i /path/to/antlr4 -p antlr -o /tmp --all -l JAVA --jdbc-
directory ~/.arcan/
```

The `--jdbc-directory` receives the path where the `arcan-db.db` file is located. In this case, by default, it is `~/.arcan/` (or `%userprofile%/.arcan` on Windows). Afterwards, you can run the dashboard and the newly-analysed project (or project version) will be available to inspect.

The CSV output

Arcan can output to file different types of information, depending on your needs and analysis options.

Arcan is able to output the following files:

- *component-metrics.csv*: a CSV file containing the list of components (units & containers, or classes and packages using Java nomenclature) parsed from the source code of your project.
    - `project` the name of the project you analysed (as provided by the `-p` parameter)
    - `versionId` the ID of the version corresponding to this component
    - `versionIndex` the index of the version corresponding to this component
    - `versionDate` the date of the version corresponding to this component
    - `name` the (full) name of the component

    - `simpleName` the simple (short) name of the component
    - `vertexLabel` the label of the component (e.g. 'unit', 'container', etc...)
    - `ComponentType` the language-specific construct type of the component (e.g. CLASS, INTERFACE, ABSTRACT_CLASS, etc..)
    - `vertexId` the ID of the component in the dependency graph of the version corresponding to this component. Can be used as a joining column together with the id (or index) of the version
    - the remainder of the columns correspond to the software metrics calculated for the components, which are, but are not limited to: `ChangeHasOccurred`, `LinesOfCode`,`filePathReal`,`IsNested`,`FanIn`,`AbstractnessMetric`,`InstabilityMetric`,`TotalAmountOfChanges`,`PageRank`,`NumberOfChildren`,`FanOut`.

- *component-membership.csv*: a CSV file that conceptually corresponds to the subgraph of the dependency graph created by all edges that represent Membership relations. This file contains the information of what container contains what component.

    - `project` the name of the project you analysed (as provided by the `-p` parameter).
    - `versionId` the ID of the version corresponding to this component.
    - `versionIndex` the index of the version corresponding to this component.
    - `versionDate` the date of the version corresponding to this component.
    - `from` the name of the component that is a child in the relationship (i.e. it is contained by the parent).
    - `fromId` the id of the component that is a child in the relationship (i.e. it is contained by the parent).
    - `childFilePathReal` the real path of the child in the file system (i.e. no links and absolute).
    - `edgeId` the id of the vertex in this version. Can be used as a joining column together with the id (or index) of the version
    - `edgeLabel` the specific type of relationship between child and parent.
    - `to` the name of the component that is a parent in the relationship (i.e. it contains the child).
    - `toId` the id of the component that is a parent in the relationship (i.e. it contains the child).
    - `parentFilePathReal` if it makes sense in the project's programming language paradigm, then it's the real path to the parent's location in the file system (no links and absolute).

- *component-dependencies.csv*: a CSV file that conceptually corresponds to the subgraph of the dependency graph created by all edges that represent dependency relations. This file contains the adjacency list of the dependencies within the system.

    - `project` the name of the project you analysed (as provided by the `-p` parameter).
    - `versionId` the ID of the version corresponding to this component.
    - `versionIndex` the index of the version corresponding to this component.
    - `versionDate` the date of the version corresponding to this component.
    - `from` the name of the component having a dependency (outgoing vertex).
    - `fromId` the unique ID of the component having a dependency (outgoing vertex).
    - `edgeId` the unique ID of the edge creating representing the dependency between `fromId` and `toId`.
    - `edgeLabel` the specific type of relationship between dependant and dependendUpon.
    - `Weight` the number of times this dependency is used in the dependant component.
    - `LocationList` the lines of code that use this dependency if dependant is a unit, or the names of the files that create that dependency if dependant is a container.

    - `to` the name of the component receiving the dependency (incoming vertex).
    - `toId` the unique ID of the component receiving the dependency (incoming vertex).

- *smell-characteristics.csv*: a CSV file that holds all the information concerning the architectural smells detected in the project.

    - `project` the name of the project you analysed (as provided by the `-p` parameter).
    - `versionId` the ID of the version corresponding to this smell.
    - `versionIndex` the index of the version corresponding to this smell.
    - `versionDate` the date of the version corresponding to this smell.
    - `vertexId` the ID of the vertex of the smell in this version of the system. This column can be used as a Join column.
    - `vertexLabel` the label of the smell (usually it's `smell` every time).
    - `smellType` the type of smell (cyclicDep, unstableDep, hubLikeDep, godComponent, etc...).
    - the remainder of the columns correspondf to the smell characteristics calculated by Arcan and include, but are not limited to `AffectedClassesRatio`,`AffectedComponentType`,`AfferentAffectedRatio`,`CentreComponent`,`Shape`,`Support`,`TotalNumberOfChanges`,`Size`,`EfferentAffectedRatio`,`LOCDensity`,`Strength`,`InstabilityGap`,`AffectedLevel`,`NumberOfEdges`, `PageRankWeighted`.
    - `affectedElements` the names of the elements affected by the smell. This column is only meant as a convenient way to have all the names of the components affected by the smell. Use `vertexId` and Join with the information in *smell-affects.csv* to get the list of affected components when doing data analysis.

- *smell-affects.csv*: a CSV file that describes what smell affects what component using their respectives IDs.

    - `project` the name of the project you analysed (as provided by the `-p` parameter).
    - `version` the ID of the version corresponding to this smell.
    - `versionIndex` the index of the version corresponding to this smell.
    - `versionDate` the date of the version corresponding to this smell.
    - `edgeId` the unique id of the edge.
    - `fromId` the ID of the smell in this version. Can be used to as a Join column.
    - `from` the smell type.
    - `edgeLabel` the exact label describing the role played by the component in the smell.
    - `toId` the ID of the component in this version. Can be used to join the column.
    - `to` a convenience column that can be used quickly identify the name of the component affected. It's recommended to not use this column during your analysis.

- *entity-tracking.csv*: a CSV file that links the multiple versions of entities (smells and components) to their unique ID that identifies them over time.

    - `project` the name of the project you analysed (as provided by the `-p` parameter).
    - `version` the ID of the version corresponding to this entity.
    - `versionIndex` the index of the version corresponding to this entity.
    - `versionDate` the date of the version corresponding to this entity.
    - `entityType` the type of the entity, namely the label of the vertex of the entity.

    - `vertexId` the ID of the entity in the version. This column can be used as a Join column with the columns that represent the IDs for smells or components (i.e. `smellId`, or `componentIdInVersion` from the files described above).
    - `vertexLabel` the label of the entity.
    - `similarityScore` the similarity score scored when the instance of this entity in this version was matched with its predecessor.
    - `entityAge` the number of consecutive versions (up until this point) that this entity was present in.

- *dependency-graph-X_Y.graphml*: a GraphML file that contains the dependency graph of the version with `versionIndex == X` and `versionId == Y`. This graph contained in this file is described below.

- *temporal-graph.graphml*: a GraphML file that contains all the information that maps different versions of the same entity (either smells or components) together. This graph contained in this file is described below.

The dependency graph

The dependency graph contains two types of elements: nodes and edges. In addition to these elements, a dependency graph also has some global variables, where boolean values are stored to keep track of the metrics computed on the graph, that are out of the scope of this doc page. Both nodes and edges have labels and properties. The labels distinguish between different classes of nodes/edges. The labels of the nodes are:

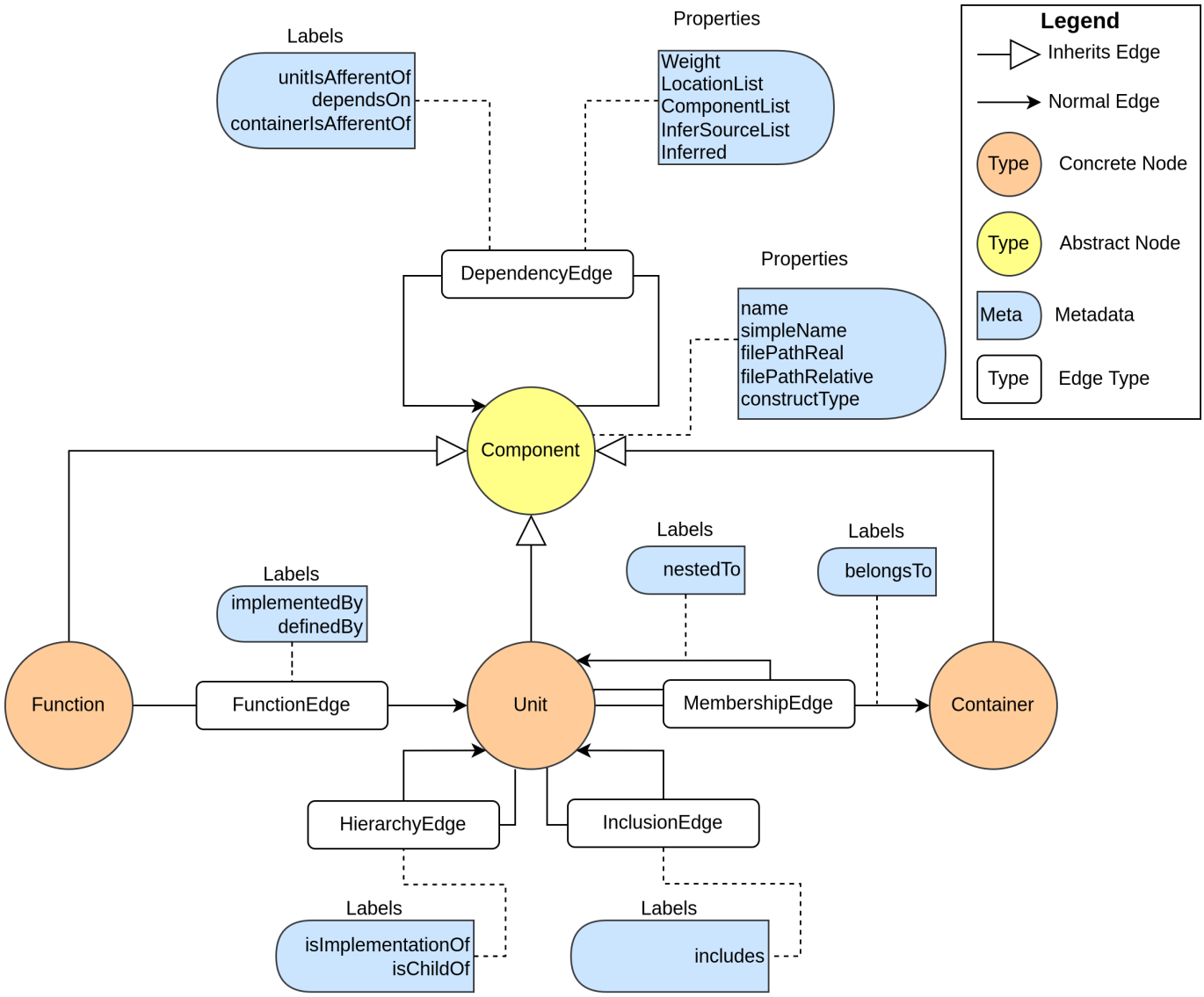
- **Component**: Is a type of label node that does actually not exist within the graph. However, Units and Containers are children of this label, and an abstraction `Component` exist in Arcan that allows you to handle this type of nodes. All Components are **uniquely** identified by their name and path.
- **Units**: Is a type of component that represents a compilation unit (e.g. a file, though not only that!).
- **Container**: Is a type of component that represents a container units or other containers.

A dedicated class exist in Arcan for each one of the nodes labels.

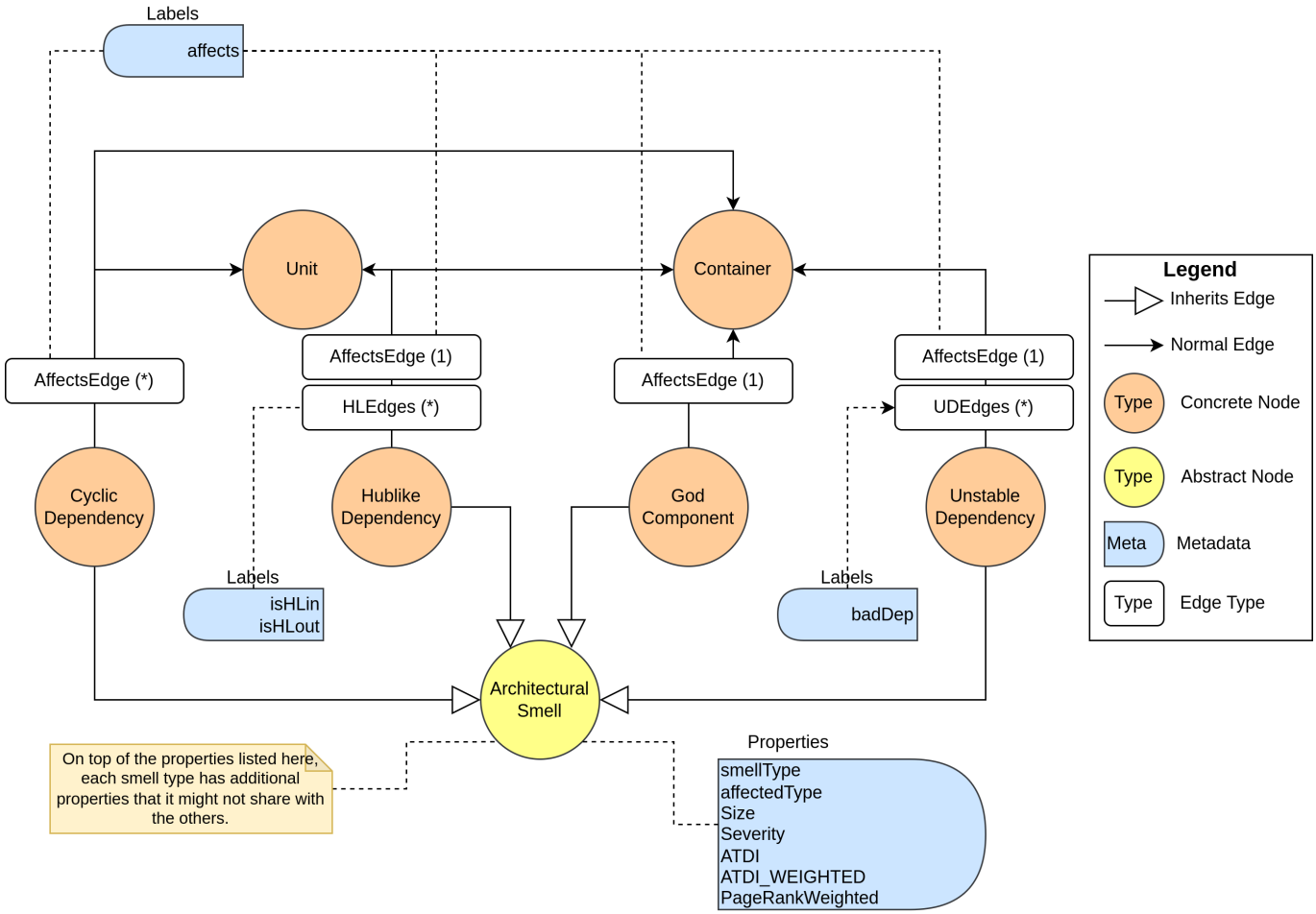
The labels of the edges are a little different though, as single class contains multiple edge labels:

- **DependencyEdges**: contains all labels representing a type of dependency. The dependency between units uses the `dependsOn` label.
- **MembershipEdges**: contains all labels representing a type of membership to another element.
- **HierarchyEdges**: contains all the labels representing a type of hierarchy between two nodes.

Both edges and nodes are able to contain properties, both custom and default, where information can be stored and accessed.



Similarly, the architectural smells also have a dedicated schema.



To visualize the depedency graph, a number of options are available:

- Arcan's own dashboard, however you need to either do all the analyses from the dashboard, or use `--jdbc-directory` option to save the state on the analysis on a DB and then run Arcan Server with that DB.
- YED Graph editor. YED is the recommended option. It provides excellent graph navigation options and is relatively fast. It is recommended to use the `Edit -> Properties Mapper` functionality for best results.
- Cytoscape. Cytoscape has great functionality, but navigation could be painful for some tasks.
- Gephi. Gephi seems to be the hardest to use, though it has a good view to explore the data stored within the graph.

The temporal graph

The temporal graph contais two main types of elements: entities and instants. Entities can either be smells or components. Entities represent instances of smells and components that are present in the system at any point in the history of the system. Instants hold information that an entity contained in a certain version of the system (denoted by the edge connecting it to the entity).

The temporal graph allows Arcan to keep track of smells and components over time, and store the values assumed by their properties at every instant in their history. For example, if our system has only two files and a smell affects both of them for the last 30 versions of the system, the temporal graph will contain 3 entity nodes (1 for each file, and 1 for the smell), 90 instant nodes (30 instants for each entity), 90 instant edges (1 for each instant node), and 60 affects edges (2 edges in each version, each going to one of the two components).

Below, you can see the schema of the temporal graph.