Archan

Release 2.0.1

Jul 30, 2018

Contents

CHAPTER 1

Archan

A Python module that analyzes your architecture strength based on Design Structure Matrix (DSM) data.

Archan is a Python module that analyzes the strength of your project architecture according to some criteria described in "The Protection of Information in Computer Systems", written by Jerome H. Saltzer and Michael D. Schroeder.

1.1 Features

- Usable directly on the command-line.
- Support for plugins. See for example the Provider plugin in dependenpy. You can also take a look at django-meerkat, a Django app using Archan.
- Configurable through command-line or configuration file (YAML format).
- Read DSM data on standard input.

1.2 Installation

Just run `pip install archan`.

1.3 Documentation

On ReadTheDocs

Archan defines three main classes: Analyzer, Provider and Checker. A provider is an object that will produce data and return it in the form of a DSM (Design Structure Matrix). The checker is an object that will analyze this DSM according to some criteria, and return a status code saying if the criteria are verified or not. An analyzer is just a combination of providers and checkers to run a analysis test suite.

1.4 Usage

1.4.1 On the command-line

Example:

```
archan -h
```

Output:

```
usage: archan [-c FILE] [-h] [-i FILE] [-l] [--no-color] [--no-config] [-v]

Analysis of your architecture strength based on DSM data

optional arguments:
  -c FILE, --config FILE  Configuration file to use.
  -h, --help              Show this help message and exit.
  -i FILE, --input FILE   Input file containing CSV data.
  -l, --list-plugins      Show the available plugins. Default: false.
  --no-color              Do not use colors. Default: false.
  --no-config             Do not load configuration from file. Default: false.
  -v, --version           Show the current version of the program and exit.
```

Other examples:

```
# Load configuration file and run archan
# See Configuration section to know how archan finds the config file
archan

# No configuration, read CSV data from file
archan --no-config --input FILE.CSV

# No configuration, read CSV data from stdin
dependenpy archan --format=csv | archan --no-config

# Specify configuration file to load
archan --config my_config.yml

# Output the list of available plugins in the current environment
archan --list-plugins
```

1.4.2 Programmatically

```
# TODO
```

1.5 Configuration

Archan applies the following methods to find the configuration file folder:

1. read the contents of the file `.configconfig` in the current directory to get the path to the configuration directory,
2. use `config` folder in the current directory if it exists,

3. use the current directory.

It then searches for a configuration file named:

1. `archan.yml`
2. `archan.yaml`
3. `.archan.yml`
4. `.archan.yaml`

Format of the configuration file is as follow:

```
analyzers: [list of strings and/or dict]
- identifier: [optional string]
  name: [string]
  description: [string]
  providers: [string or list]
  - provider.Name: [as string or dict]
      provider_arguments: as key value pairs
  checkers: [string or list]
  - checker.Name: [as string or dict]
      checker_arguments: as key value pairs
```

It means you can write:

```
analyzers:
# a first analyzer with one provider and several checker
- name: My first analyzer
  description: Optional description
  providers: just.UseThisProvider
  checkers:
  - and.ThisChecker
  - and.ThisOtherChecker:
      which: has
      some: arguments
# a second analyzer with several providers and one checker
- name: My second analyzer
  providers:
  - use.ThisProvider
  checkers: and.ThisChecker
# a third analyzer, using its name directly
- some.Analyzer
```

Every checker support an `ignore` argument, set to True or False (default). If set to True, the check will not make the test suit fail.

You can reuse the same providers and checkers in different analyzers, they will be instantiated as different objects and won't interfere between each other.

As an example, see Archan's own configuration file.

To get the list of available plugins in your current environment, run `archan --list-plugins` or `archan -l`.

1.6 Writing a plugin

1.6.1 Plugin discovery

You can write three types of plugins: analyzers, providers and checkers. Your plugin does not need to be in an installable package. All it needs to be summoned is to be available in your current Python path. However, if you want it to be automatically discovered by Archan, you will have to make it installable, through pip or simply `python setup.py install` command or equivalent.

If you decide to write a Python package for your plugin, I recommend you to name it `archan-your-plugin` for consistency. If you plan to make it live along other code in an already existing package, just leave the name as it is.

To make your plugin discoverable by Archan, use the `archan` entry point in your `setup.py`:

```
from setuptools import setup

setup(
    ...,
    'entry_points': {
        'archan': [
            'mypackage.MyPlugin = mypackage.mymodule:MyPlugin',
        ]
    }
```

The name of the entry point should by convention be composed of the name of your package in lower case, a dot, and the name of the Python class, though you can name it whatever you want. Remember that this name will be the one used in the configuration file.

Also a good thing is to make the plugin importable thanks to its name only:

```
import mypackage.MyPlugin
```

But again, this is just a convention.

1.6.2 Plugin class

You can write three types of plugins: analyzers, providers and checkers. For each of them, you have to inherit from its corresponding class:

```
from archan import Analyzer, Provider, Checker

class MyAnalyzer(Analyzer): ...
class MyProvider(Provider): ...
class MyChecker(Checker): ...
```

A provider or checker plugin must have the following class attributes:

- identifier: the identifier of the plugin. It must be the same name as in your entry points, so that displaying its help tells how to summon it.
- name: the verbose name of the plugin.
- description: a description to explain what it does.
- (optional) arguments: a tuple/list of Argument instances. This one is only used to display some help for the plugin. An argument is composed of a name, a type, a description and a default value.

```
from archan import Provider, Argument

class MyProvider(Provider):
    identifier = 'mypackage.MyProvider'
    name = 'This is my Provider'
    description = """
    Don't hesitate to use multi-line strings as the lines will be de-indented,
    concatenated again and wrapped to match the console width.

    Blank lines will be kept though, so the above line will not be removed.
    """

    arguments = (
        Argument('my_arg', int, 'This argument is useful.', 42),
        # don't forget the ending comma if you have just one   ^   argument
    )
```

Additionally, a checker plugin should have the `hint` class attribute (string). The hint describe what you should do if the check fails.

For now, the analyzers plugins just have the `providers` and `checkers` class attributes.

1.6.3 Plugin methods

A provider must implement the `get_dsm(self, **kwargs)` method. This method must return an instance of `DSM`. A DSM is composed of a two-dimensions array, the matrix, a list of strings, the keys or names for each line/column of the matrix, and optionally the categories for each key (a list of same size).

```
from archan import DSM, Provider

class MyProvider(Provider):
    name = 'mypackage.MyProvider'

    def get_dsm(self, my_arg=42, **kwargs):
        # this is where you compute your stuff
        matrix_data = [...]
        entities = [...]
        categories = [...] or None
        # and return a DSM instance
        return DSM(matrix_data, entities, categories)
```

A checker must implement the `check(self, dsm, **kwargs)` method.

```
from archan import DSM, Checker

class MyChecker(Checker):
    name = 'mypackage.MyChecker'

    def check(self, dsm, **kwargs):
        # this is where you check your stuff
        # with dsm.data, dsm.entities, dsm.categories, dsm.size (rows, columns)
        ...
        # and return True, False, or a constant from Checker: PASSED or FAILED
        # with an optional message
        return Checker.FAILED, 'too much issues in module XXX'
```

1.6.4 Logging messages

Each plugin instance has a `logger` attribute available. Use it to log messages with `self.logger.debug`, `info`, `warning`, `error` or `critical`.

1.7 Available plugins

Here is the list of plugins available in other packages.

1.7.1 Providers

- `dependenpy.InternalDependencies`: Provide matrix data about internal dependencies in a set of packages. Install it with `pip install dependenpy`.

1.8 License

Software licensed under ISC license.

1.9 Development

To run all the tests: `tox`

CHAPTER 2

Reference

CHAPTER 3

Roadmap

Archan currently checks for criteria that are only applicable on DSM representing either internal dependencies of an application, or access control rights.

In fact, it is possible to represent many different things with DSM. Also, each criterion does not apply to each of these different things. Hence, we can build the following matrix, with the different types of DSM data, and the different criteria to check against the DSMs.

Criterion \ DSM data	Code source	Access control	Internal dependencies	Project dependencies	Network architecture	Risks
Code clean	■					
Least privileges		■				
Separation of privileges		■				
Complete Mediation			■			
Layered architecture			■	■	■	
Economy of mechanism			■	■	■	
Least common mechanism		■	■	■	■	
Open Design	■	■	■	■	■	■

The arrow going from left to right in the header represents the scale growing up: from the code source to dependency data to network architecture to risk analysis.

The blue cells indicate that the criterion can be run on the related type of DSM. As you can see, Open Design criterion can be checked against any type of DSM, as it is just a matter of telling if yes or no the design of the entity is open. At the contrary, Code Clean criterion is only applicable on code source. Least Privileges and Separation of Privileges criteria are only applicable on access control data since it requires roles and privileges data to be checked (software dependencies do not offer such data).

However, everything in this matrix is not fixed. We could maybe imagine running a Complete Mediation check on the network architecture or a Layered Architecture check on access control data. The only limit is imagination: what data to present, how to generate them, how to check them.

The risk column is a perfect example: we could define risks for pretty much each kind of entity: code source, access control, network architecture..., but against which criteria can we check them? This has yet to be specified.

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.2 Documentation improvements

Archan could always use more documentation, whether as part of the official Archan docs, in docstrings, or even on the web in blog posts, articles, and such.

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/Genida/archan/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

4.4 Development

To set up *archan* for local development:

1. Fork archan (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/archan.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the tests with one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

[1] If you don't have all the necessary python versions available locally you can rely on...
- **Travis**: it will run the tests for each change you add in the pull request. It will be slower though...
- **pyenv**:

```
# important libraries to compile Python
sudo apt install -y libssl-dev openssl zlib1g-dev sqlite3 libsqlite3-dev libbz2-dev bzip2

git clone https://github.com/pyenv/pyenv.git ~/.pyenv
export PATH="${HOME}/.pyenv/bin:${PATH}"
eval "$(pyenv init -)"

pyenv install 3.5.3
pyenv install 3.6.0  # etc.
pyenv global system 3.5.3 3.6.0
```

4.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

CHAPTER 5

Authors

- Pierre Parrend
- Timothée Mazzucotelli
- Florent Colin
- Jean-Louis Mandel

CHAPTER 6

Changelog

6.1 2.0.1 (2017-06-27)

- Fix usage of DSM size.
- Start implementing other concepts (MDM and DMM).

6.2 2.0.0 (2017-06-26)

- Change license from MPL 2.0 to ISC.
- Add command-line tool.
- Rewrite architecture to support plugins.

6.3 1.0.0 (2016-10-06)

- Add documentation.

6.4 0.1.0 (2016-10-06)

- Alpha release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search