

CALint: A Tool for Enforcing the Clean Architecture's Dependency Rule in Python

Clevio Orlando de Oliveira Junior[1], Jonathan Carvalho[1], Fábio Fagundes Silveira[1(✉)], Tiago Silva da Silva[1], and Eduardo Martins Guerra[2]

[1] Federal University of São Paulo – UNIFESP, São José dos Campos, São Paulo, Brazil
{fsilveira,silvadasilva}@unifesp.br
[2] Free University of Bolzen-Bolzano – UNIBZ, Bozen-Bolzano, Italy

Abstract. Clean Architecture (CA) aims to address the need for more loosely coupled components and better cohesion. CA focuses on preparing software engineers to write more stable, durable, and flexible applications capable of distinguishing between details (e.g., what framework it uses) and the business logic requirements. A literature review shows that considerable effort has been devoted to cataloging and solving code smells related to code, often called code smells. However, the same does not apply to architecture smells – its software architecture counterpart. Similar research regarding other programming languages such as Java, PHP, or C# represents noteworthy works in the area, but they do not address Python applications directly. This work directs efforts towards redesigning and adapting existing Python programs to the CA principles by detecting the code smells that break the CA constraints through the developed CALint tool. Moreover, this approach proposes two extended refactoring techniques to solve these smells efficiently by grouping and comparing static code analysis and reuse them to enforce Clean Architecture's Dependency Rule programmatically. To demonstrate the feasibility of the two refactoring techniques described in this work and the CALint tool, we applied them to three different case studies. The major findings of this work include two extended refactoring techniques and the development of a tool to verify non-conformities related to the Clean Architecture dependency rule. The results show common cases where the dependency rule was violated and highlighted by the CALint tool, which are fixed with the support of refactoring steps.

Keywords: Software engineering · Clean architecture · Code smells · Refactoring

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
O. Gervasi et al. (Eds.): ICCSA 2022 Workshops, LNCS 13381, pp. 534–549, 2022.
https://doi.org/10.1007/978-3-031-10548-7_39

1 Introduction

Refactoring techniques are constantly needed in the software development process. In fact, in multidisciplinary software projects, it is expected that due to tight deadlines software developers choose shortcuts, which result in trade-offs that allow the customer's expectations to be met in the short term, accumulating technical debts that inflate the long-term maintenance cost of an application [11].

For a long time, refactoring meant hiding the parts of the code that dealt with input and output through subroutines. Nowadays, one can see that refactoring means something different. This time closer to exposing the *I/O* operations and hiding the operations that deal with the business logic [20]. Among the methods that allow such detection, is it possible to cite the analysis of the program dependency graph [5]. This directed graph represents a dependency between elements of a program and can be used, for example, to detect duplicate codes [13], which configures a code smell proposed by [8].

Currently, the maintenance benefits that architectural styles such as the Clean Architecture or the Hexagonal Architecture [4] provide are widely discussed. However, there is a lack of discussions about adapting existing programs to a specific architecture or even detecting violations of their fundamental constraints.

This article proposes two code refactoring techniques to adapt existing Python codes to the Clean Architecture constraints. Furthermore, it shows a tool named *CAList*, developed to automate this process, responsible for creating a set of rules, based on existing static code analysis tools, capable of identifying and warning about smells that violate the dependency rule Clean Architecture principle in Python applications. Although our proposed tool and refactoring techniques are an ongoing research project, preliminary results show that they are feasible when applied to a few selected case studies from the Python developer community.

The remaining of this paper is structured as follows: Sect. 2 presents an overview of current work involving architecture smells detecting and refactoring techniques for the Clean Architecture. Section 3 describes the tool structure and Sect. 4 presents two extended refactoring techniques proposed in this work. Section 5 presents the application of the tool in three different case studies, aiming to demonstrate the feasibility of our approach. Finally, Sect. 6 concludes the paper and points out future works.

2 Related Works

Garcia et al. [9] describe what architecture smells are and compare them with other code smells. They list and define new architectures smells, citing examples in industrial systems and showing these smells with diagrams, possible causes, and tradeoffs. This paper also makes efforts to catalog architecture smells. However, differently from the work proposed by Garcia et al. [9], we focus only in the architecture smells related to the Clean Architecture. Further, we propose explanations and solutions for refactoring them with a programmatic approach.

Bui [2] proposes refactoring techniques for Clean Architecture but in a different context. First, the author explains various design patterns and architectural

models, as well as mentions principles like the dependency rule, but does not have a focus on code smells and refactoring techniques. Also, several good practices are mentioned as well as their reasons within the context of Android and reactive programming applications.

The work of Velasco-Elizondo et al. [22] has the goal of proposing smells and refactoring techniques for a specific architectural pattern. However, as the Clean Architecture has more layers and rules than the MVC one, the techniques for finding and solving smells need to be adapted.

A refactoring method for architecture smells, more specifically, the *Cyclic Dependency Smell* is proposed by Rizzi et al. [21]. They also extend functions of an existing application based on graphs (*Arcan* [6]), and perform an experiment in a large base of programs. However, the work does not focus on Python language neither the CA's specific rules.

Beltrão et al. [1] present a project called "*Project Case*". The aim is to reduce technical debt (TD), a metaphor in which developers take shortcuts during software development and negatively impact quality, completeness, or safety. The work implements technical debt detection rules through a plugin developed to the *SonarQube*[1], a continuous code quality inspection tool based on static analysis.

Even though there are important related works in the literature, neither of them focuses on analyzing and justifying these code smells nor focuses on the Python programming language.

3 The CALint Tool

3.1 General View

CALint is a tool developed to check if a given Python project complies with the Clean Architecture dependency rules and, if not, to point out the places where the non-conformities were detected. The objectives of this tool are:

- Use the own proposed tool as an object of study, pointing out during development the times when the dependency rule was violated, in a process analogous to *bootstrapping*;
- Facilitate the configuration of other existing tools on the market to efficiently establish rules shaped for the Clean Architecture. Besides, to accelerate the process of development and refactoring of existing programs to the constraints of the Clean Architecture, more specifically, the dependency rule one;
- Start a development project where checking for dependency rules becomes continuous and simplified through the whole software life cycle, integrating it with tools used in the market;
- Create an approach that follows and/or checks the Clean Architecture constraints differently from those commonly found, which primarily focus on data reading and storing operations.

[1] https://www.sonarqube.org.

3.2 Architecture and Development

As pointed out by Martin [15], the folder structure should reveal the features of the program and not be strictly framework-oriented or driven to *I/O* mechanisms. This project adheres to this premise regarding its organization. Figure 1 depicts the folders and location of its architectural layers.

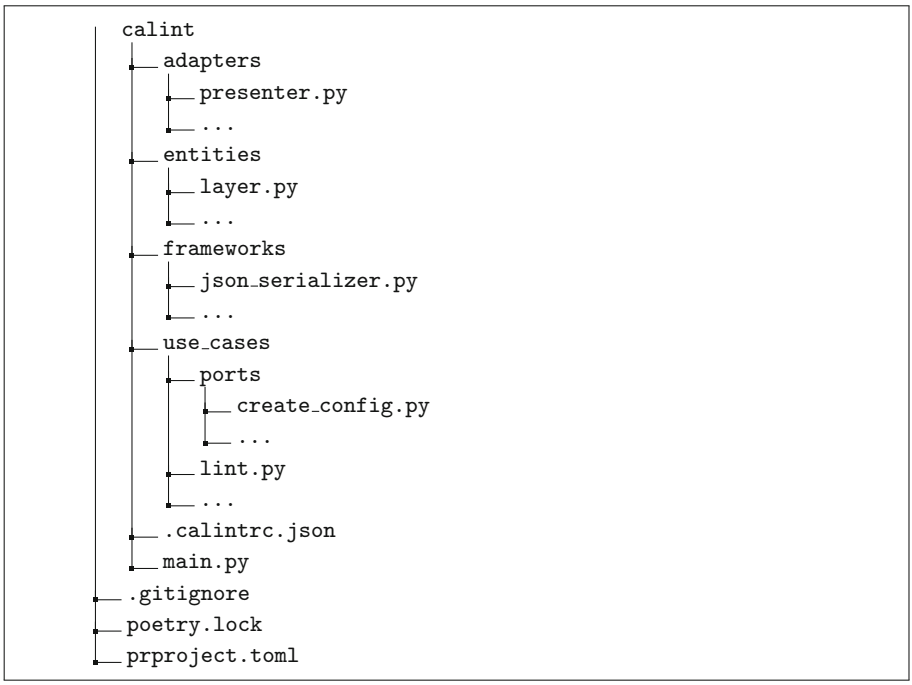

Fig. 1. CALint folder structure

CALint has five architectural layers: main, frameworks, adapters, use_cases, and entities (Fig. 2). There is an extra layer, named Main. It is the gateway to the application. The function that calls the use cases in order, sending parameters and injecting dependencies on each one is in the Main. First, it calls the use case responsible for configuring the entities create_config, then the use case that creates the results (lint).

The following frameworks are used in the CALinst:

- Click (click_printer.py): responsible for formating and coloring the output from the command-line interface.
- Import Linter (importlinter_linter.py): the main framework in the project, it analyzes the dependencies between layers of the application based on the rules specified by the user.
- json (json_serializer.py), despite being a standard library provided by Python, it is crucial that the programmer has the freedom to change it and choose

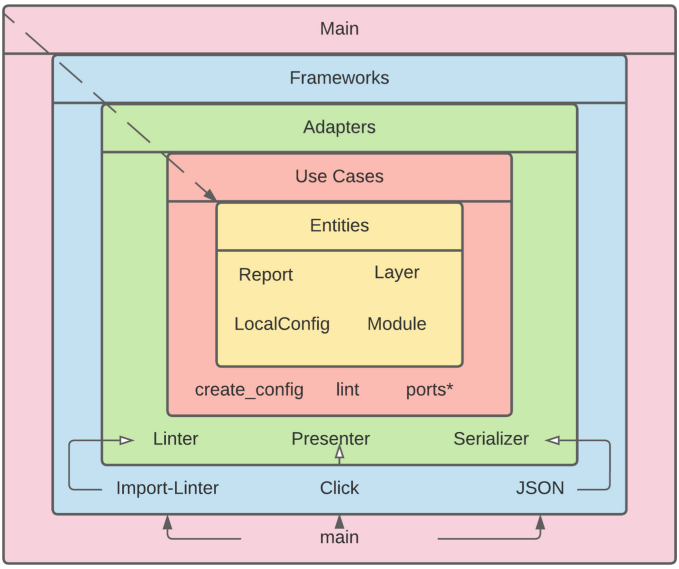

Fig. 2. CALint dependency architecture

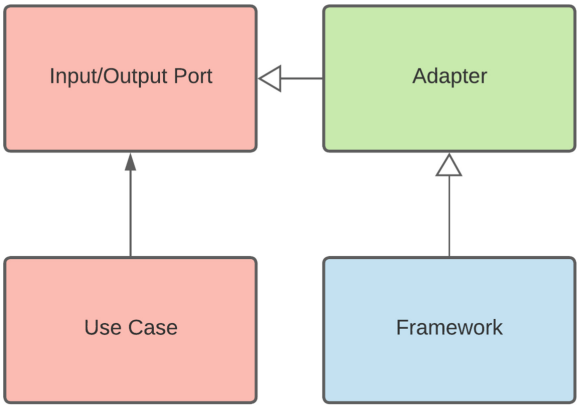

Fig. 3. Flow of dependencies from the use case to frameworks

alternatives of data serializers in the project. After all, serializing is an I/O process. This way, if there is a requirement to use another serializer framework (e.g. a serializer for `toml`, an alternative library of `json`), it can be changed easily, due to dependency injection [19].

The adapters are the project's adapters, both input, and output. An adapter can have ports for one or more input or output use cases. In this case, they make the connections between the libraries and the use cases through the implementation of the use case ports (Fig. 3). The CALint adapters are:

- LinterAdapter: it implements LintPort. This adapter implements the linter output method, calling an abstract method, later implemented by one of the classes that use the frameworks, called lint. It receives a configuration of type LocalConfig and returns a Report.
- PresenterAdapter, it implements the PrinterPort. This adapter implements the call to present passing the contents of the Report so that the responsible framework presents the contents of the Report.
- SerializerAdapter, it implements CreateConfigPort, calling the deserialize. In addition, SerializerAdapter also provides the signature of serialize, which, unlike deserialize, takes a dictionary and returns a string.

Use cases coordinate the application's business logic and populate and regulate entity data required by the application. They do not know about adapters, as they define the input and output ports (abstract classes). In this way, adapters that implement the same ports can be inserted or removed as needed without knowledge of the use cases. There are three ports (Fig. 4):

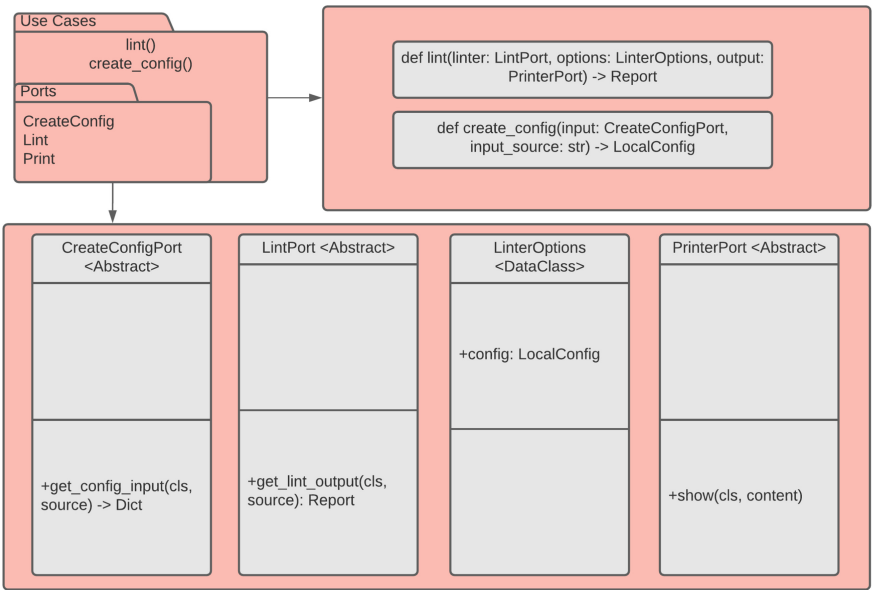

Fig. 4. CALint use cases and ports

- LintPort: responsible for defining the signature of the output after executing the linter. Defines that the configuration must be of type LocalConfig and the return must be an entity, Report.
- PrinterPort: responsible for defining the signature of the method that displays the return contents, regardless of the program's output interface. Requires that it must have a name with only one dynamic typed argument and must return nothing.
- CreateConfigPort: responsible for defining the architecture of an abstract method and receiving an arbitrary source, returning a dictionary (Dict).

Entities describe the enterprise business or the business objects of an application. They also represent objects and instructions and how to handle them. This part of the system must be unaware of which technologies implementation are accessing them. The main entities of CALint are:

- Layer: represents an architectural layer. Stores the priority data, the physical path of the import, and the virtual name of the Layer.
- LocalConfig: represents the local configuration, that is, the layers and the root of the project to be analyzed.
- Module: represents a module, contains the physical path and Layer to which this module belongs.
- BrokenRule: represents a broken rule. Contains the start and the end modules and the line where the nonconformity is found.
- Report: represents a report, that is, the listing of the broken rules.

3.3 Setting up the CALint

The tool is configured with the JSON file .calintrc.json, which must be placed in the project's root folder. Within this configuration file, it is necessary to have at least the following entries: roots and the subsequent layers' names (Fig. 5).

```
{
    "roots": ["root_folder_one", "root_folder_two"],
    "frameworks": ["root_folder_two.frameworks", 0],
    "adapters": ["root_folder_two.adapters", 1],
    "use-cases": ["root_folder_one.use_cases", 2],
    "entities": ["root_folder_one.entities", 3]
}
```

Fig. 5. The CALint JSON configuration file

Here, every key is a string (as per JSON requirements), and every value consists of an array. The array will always contain one string and one integer (works like a tuple), except when the key is the root one. Roots must be an array of strings where every string is a root directory, and this is especially useful when there is no "main" folder and folders are scattered around the root folder.

For every other possible key-value pair, the key represents the virtual name of an architectural layer (e.g., "use cases"). The value will contain the layer's module path (e.g., `root_folder_one.use_cases`) and an integer value, representing how external that architectural layer is. The numbering sequence must start at "0" (the outermost layer), while the innermost layer should have the highest number (e.g., "3").

4 Proposed Refactoring Techniques

This section describes two proposed refactoring techniques applied to pave the way for existing Python code to adhere to the CA dependency rule.

4.1 Extract Port – Adapting the Existing Source Code

In order to refactor the non-conformities found to adhere to the CA's dependency rules, it is necessary to know where a module should be located in the application and which modules it is allowed to import.

During the case studies, we noticed that developers commonly forget that adapters should not be called directly from the use cases or entity layer. By doing that, use cases and adapters get tightly coupled to each other. In order to solve this issue, it is necessary to refactor the code so that the adapters become use case-dependent.

Consider the architecture smell represented in Fig. 6. The solution applied to this case consists of the following steps:

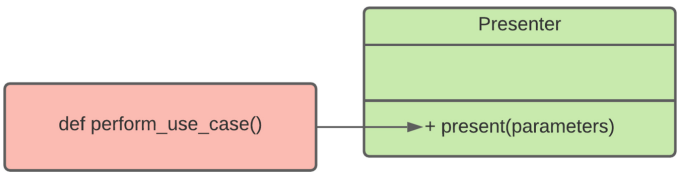

Fig. 6. Violating the dependency rule – a scenario where a use case is dependent on an adapter

1. Check what exactly the use case output needs to be and what is available to pass as a parameter;
2. Verify if such a situation is an input or output operation;
3. Create a use case centric function signature, specifying the parameters as well as the returning values;

4. Extract this information as an interface or as an abstract class (*Extract Interface* [8]), making the adapter dependent on or inheriting from this new class;
5. Replace adapter function calls with use cases;
6. Inject dependencies.

In the Sect. 5.1 we describe an application of this refactoring technique to handle non-conformities reported by CALint.

4.2 Extract Repository – Refactoring Code that Accesses Storage

As stated by Cockburn [4], there are primarily two types of adapters: primary and secondary. This nomenclature exemplifies that some adapters tend to be more responsible for the input of a program, whereas others are more for the output. According to the author, there are three technical solutions (which depend on the programming language): i) define interfaces and inject dependencies; ii) inject anonymous functions (also known as lambda expressions), and iii) inject objects that have common signatures (*duck typing* and dynamic typing [17]).

A program that manages repositories or data storage services, option "i" [4] can commonly be used to perform refactoring of programs that depend directly on an ORM. The ORM or *Object Relational Mapping* refers to a type of framework that tries to abstract and encapsulate the database code, mapping the data present in the database to objects of a programming language [3]. However, ORMs are nothing more than frameworks. In order to keep them flexible and allow replacement, keeping the decoupling between the application and the database, *Repository Patterns* [16,18] is commonly used, as shown in the Python Clean Architecture example (Sect. 5).

When a program directly accesses the ORM, the following best practices must be followed:

1. Group the program accesses (calls) by the database table name. Since not necessarily all entity's data will be recorded, storing all entity data would not only cause the entity to become coupled to the outside world but also expose sensitive information;
2. Create an interface using the *Repository Pattern* for every database table. Group typical operations within the generic repository, such as searching, creating, deleting, updating, and sorting;
3. Make the generic repository logic implemented concretely within each framework. In the composition root, frameworks are then injected. So, the repository works as a gateway to one or more use cases.

Figure 7 illustrates this situation and shows a possible solution by using this refactoring technique. It is quite common to call frameworks in the use case layer, especially when the use case itself deals with CRUD operations. Nevertheless, that violates the CA dependency rule, as shown in the source code example

(Fig. 7, line 7), since there is a calling to the *SQLAlchemy* directly from the use case. Although a repository can be seen as a use case port, the database contents or wherever that repository host is not part of the use case. They are implementation details.

```
1         from config import session
2         from models import User
3
4         def list_users():
5             # CODE SMELL! Use Case calling the
6             # SQLAlchemy ORM framework directly...
7             return session.query(User).all()
```

```
8         # Snippet code #1
9         # on usecases/ports
10        class UserRepository(ABC):
11            @staticmethod
12            @abstractmethod
13            def all():
14                pass
15
16        # Snippet code #2
17        # on frameworks/DB
18        # stuff directly related to the db
19        from sqlalchemy import create_engine
20
21        from .config import session
22        from .models import User
23
24        class DBUserRepository(UserRepository):
25            @staticmethod
26            def all():
27                return session.query(User).all()
28
29        # Snippet code #3
30        # on usecases
31        def list_users(repository: UserRepository):
32            return repository.all()
33
34        # now when calling list_users, somewhere else
35        list_users(DBUserRepository())  # here we inject the repository as a
                                            dependency
```

Fig. 7. Extract repository refactoring technique example

To solve this code smell, one can create a repository (e.g., UserRepository) to host the function signatures as an abstract class (line 10). Then, a concrete class (e.g., DBUserRepository) provides the details to the repository in the framework layer (lines 24–27). From now on, the use case only knows about UserRepository (line 31).

5 Case Studies and Result Analysis

To demonstrate the feasibility of the CALint tool and the two refactoring techniques described in this work, we applied them to three different case studies.

5.1 Bootstraping the CALint Itself

One practice in computer science is the *bootstrapping* [14]. Bootstrapping is the technique used by a newly created compiler to compile itself. In other words, the idea is to produce a self-compiling compiler.

Thus, we applied this technique to guide and verify whether the dependency rule is achieved or not during the development of the own CALint tool. The configuration JSON file of dependencies is depicted in Fig. 8.

```
{
    "roots": ["calint"],
    "main": ["calint.main", 0],
    "frameworks": ["calint.frameworks", 1],
    "adapters": ["calint.adapters", 2],
    "use_cases": ["calint.use_cases", 3],
    "entities": ["calint.entities", 4]
}
```

Fig. 8. The CALint JSON dependency file configuration for the CALint verification

With this structure, the next subsections describe times when there were non-conformities.

In one of the tool's first versions, as depicted in Fig. 9, it was possible to run the program and figure out that the use cases did not create a port, but rather the adapter was driving the development of business logic, a visible violation of the dependency rule and therefore a code smell according to the Clean Architecture constraints.

A new port was created in the use cases as an abstract class to refactor this code. The adapter had to inherit to be called by the use case after injecting dependencies, thus causing the adapter to depend on the instructions provided by the use case, not the other way around.

An example of such a situation can be seen in the snippet code of Fig. 10, where a use case imports an adapter. In the lint method, the first parameter is the adapter itself, not a port. Thus, to make it compliant with the clean architecture dependency rules and address this smell, we need to extract the port and use it as the first parameter of the lint use case method.

A similar problem was detected in one of the latest versions of the CALint tool. At this time, a use case layer was directly importing a presenter from the adapter layer (Fig. 11). Again, this situation was fixed by using the same refactoring technique. Due to the lack of space, reduced reports from CALint are depicted from now on.

```
/home/me/ca-lint/calint/.calint/importlinter.ini
=============
Import Linter
=============

---------
Contracts
---------

Analyzed 27 files, 40 dependencies.
-----------------------------------

Clean Architecture Layers BROKEN

Contracts: 0 kept, 1 broken.


----------------
Broken contracts
----------------

Clean Architecture Layers
-------------------------

calint.use_cases is not allowed to import calint.adapters:

- calint.use_cases.lint -> calint.adapters.linter (l.1)


➜  calint git:(main) ✗
```

Fig. 9. Example of a dependency rule-breaking problem identified by CALint

```
1    from calint.adapters.linter import LinterAdapter
2    from calint.use_cases.ports import LinterOptions
3
4    def lint(linter: LinterAdapter, options: LinterOptions):
5        return linter.lint(options.config)
```

```
1
2    from calint.use_cases.ports import LinterOptions, LintPort
3
4    def lint(linter: LintPort, options: LinterOptions):
5        return linter.lint(options.config)
```

Fig. 10. Extract port refactoring technique example

```
use_cases (calint.use_cases.lint) imported adapters (calint.adapters.presenter) at line
 1
1 rule broken
(.venv) ➜  calint git:(main) ✗ poetry run python cli.py
- use_cases (calint.use_cases.lint) imported adapters (calint.adapters.presenter) at li
ne 1

1 rule broken
(.venv) ➜  calint git:(main) ✗
```

Fig. 11. CALint detecting a non-conformity in the CALint architecture design: the use case layer importing a presenter from the adapter layer

5.2 *Rent-o-Matic* Project Case Study

Rent-o-Matic is a demo project[2], comprising a companion repository to [10]. As the physical files tend to differ from project to project, first, we need to conduct a pre-processing step to create the files __init__.py, need by the Python language to solve the modules problem import issue [7]. Shortly, it was necessary to restructure some folders of the application as well as create the JSON dependency rule file (Fig. 12).

```
{
    "roots": ["rentomatic"],
    "main": ["rentomatic.app", 0],
    "frameworks": ["rentomatic.rest", 1],
    "adapters": ["rentomatic.repository", 2],
    "use-cases": ["rentomatic.use_cases", 3],
    "entities": ["rentomatic.domain", 4]
}
```

Fig. 12. Rent-o-Matic CALint JSON configuration file

An empty abstract class was also detected, where its only purpose was to check if something was part of the Rent-o-Matic domain. However, this class was in the use cases folder, and CALint detected it correctly, as shown in Fig. 13. To fix this nonconformity caused by the relocation of the folders, the tool just moves it to the entities folder.

```
→  rentomatic git:(master) ✗ calint
- entities (rentomatic.domain.storageroom) imported use-cases (rentomatic.use_cases.sha
red.domain_model) at line 1

1 rule broken
```

Fig. 13. CALint detecting a non-conformity in the *Rent-O-Matic* design: the domain layer importing a class from the use case layer

5.3 Python Clean Architecture Example Case Study

The *Python Clean Architecture Example* [12][3] is an open source project on *GitHub* with the aim of illustrating the Clean Architecture in a *Flask* application.

Considering the study case architecture diagram (Fig. 14), we set the JSON dependency rules file (Fig. 15), creating the folder called frameworks with the modules main and in_memory_post_repository. In addition to that, the interfaces folder moved into the use cases folder because this folder represents the ports. Since CALint did not detect any problem, no code changes were required.

[2] https://github.com/lgiordani/rentomatic.
[3] https://github.com/heumsi/python-clean-architecture-example.

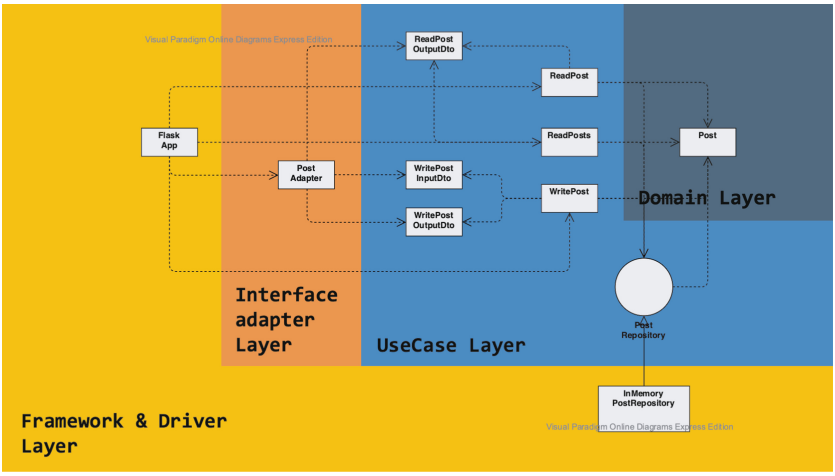

Fig. 14. Python CA example architecture diagram [12].

```
{
    "roots": ["frameworks", "domain", "adapter"],
    "frameworks": ["frameworks", 0],
    "adapter": ["adapter", 1],
    "use_cases": ["domain.use_case", 2],
    "entities": ["domain.entity", 3]
}
```

Fig. 15. The Python Clean Architecture example JSON dependency configuration file

As shown by Fig. 16, CALint reports no nonconformities regarding the CA dependency rule in this case study.

```
→  python-clean-architecture-example git:(master) ✗ calint
No rules broken.
```

Fig. 16. CALint not highlighting any nonconformities

6 Conclusion and Future Work

Clean Architecture defines rigid communication layers by centering the program on its domain rather than its composing frameworks. As a result, it is possible to make the program completely configurable, as the details become flexible enough to be rewritten and plugged in on demand.

However, it is possible to refactor and adapt existing programs to the Clean Architecture since most refactorings involve principles well established in the literature, such as dependency inversion and dependency injection. Experiments

in the industry are still necessary to verify and evaluate the time spent in refactor systems with large code bases.

The work herein presented focuses on exploring the possibility of refactoring existing programs to generate step-by-step instructions for more common architecture smells. Hence, the significant findings of this paper include two extended refactoring techniques and the development of a tool to verify non-conformities with the Clean Architecture dependency rule.

Our approach results show common cases where the dependency rule was violated. The CALint tool highlighted these cases, which, in turn, enabled the correction with the support of the refactoring steps.

Regarding future works, some extensions can be added to the approach to improve the tool operation, such as, add new functionalities, and integrate it into industrial software development. Finally, improving the command-line interface, implementing the invalid dependency rule configuration checking, and supporting external packages are part of our next steps.

Acknowledgments. The authors would like to thank the São Paulo Research Foundation (FAPESP) grant 2018/22064-4 – for financial support.

References

1. Beltrão, A., Farzat, F., Travassos, G.: Technical debt: a clean architecture implementation. In: Anais Estendidos do XI Congresso Brasileiro de Software: Teoria e Prática. SBC, Porto Alegre, RS, Brasil, pp. 131–134 (2020). https://doi.org/10.5753/cbsoft_estendido.2020.14620. https://sol.sbc.org.br/index.php/cbsoft_estendido/article/view/14620
2. Bui, D.: Reactive programming and clean architecture in Android development. Metropolia Ammattikorkeakoulu (2017)
3. Chen, T.H., et al.: An empirical study on the practice of maintaining object-relational mapping code in Java systems. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), pp. 165–176. IEEE (2016)
4. Cockburn, A.: Hexagonal architecture (2005). https://alistair.cockburn.us/hexagonal-architecture/
5. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. (TOPLAS) **9**(3), 319–349 (1987)
6. Fontana, F.A., Pigazzini, I., Roveda, R., Tamburri, D., Zanoni, M., Di Nitto, E.: Arcan: a tool for architectural smells detection. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 282–285. IEEE (2017)
7. Python Software Foundation: Python language reference, modules (2022). https://docs.python.org/3/tutorial/modules.html
8. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
9. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Toward a catalogue of architectural bad smells. In: Mirandola, R., Gorton, I., Hofmeister, C. (eds.) QoSA 2009. LNCS, vol. 5581, pp. 146–162. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02351-4_10

10. Giordani, L.: Clean Architectures in Python, 2nd edn. Leanpub (2018). https://leanpub.com/clean-architectures-in-python
11. Guo, Y., et al.: Tracking technical debt - an exploratory case study. In: ICSM 2011: Proceedings of the 2011 27th IEEE International Conference on Software Maintenance, pp. 528–531, September 2011. https://doi.org/10.1109/ICSM.2011.6080824
12. Heumsi: Python clean architecture example (2020). https://github.com/heumsi/python-clean-architecture-example
13. Hotta, K., Higo, Y., Kusumoto, S.: Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In: 2012 16th European Conference on Software Maintenance and Reengineering, pp. 53–62. IEEE (2012)
14. Louden, K.C.: Compiler Construction. Cengage Learning, Boston (1997)
15. Martin, R.C.: Clean Architecture: A Craftsman's Guide to Software Structure and Design. Robert C. Martin Series. Prentice Hall, Boston (2017). https://www.safaribooksonline.com/library/view/clean-architecture-a/9780134494272/
16. Microsoft:net microservices - architecture e-book (2021). https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design
17. Milojkovic, N., Ghafari, M., Nierstrasz, O.: It's duck (typing) season! In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 312–315. IEEE (2017)
18. Prajapati, M., et al.: ASP.NET MVC-generic repository pattern and unit of work. Int. J. All Res. Writ. **1**(1), 23–30 (2019)
19. Razina, E., Janzen, D.S.: Effects of dependency injection on maintainability. In: Proceedings of the 11th IASTED International Conference on Software Engineering and Applications, Cambridge, MA, p. 7 (2007)
20. Rhodes, B.: The clean architecture in Python (2014). https://archive.org/details/pyvideo_2840___The_Clean_Architecture_in_Python
21. Rizzi, L., Fontana, F.A., Roveda, R.: Support for architectural smell refactoring. In: Proceedings of the 2nd International Workshop on Refactoring, pp. 7–10 (2018)
22. Velasco-Elizondo, P., Castañeda-Calvillo, L., García-Fernandez, A., Vazquez-Reyes, S.: Towards detecting MVC architectural smells. In: Mejia, J., Muñoz, M., Rocha, Á., Quiñonez, Y., Calvo-Manzano, J. (eds.) CIMPS 2017. AISC, vol. 688, pp. 251–260. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-69341-5_23