Clevio Orlando de Oliveira Junior

Uma Ferramenta para a Detecção de Inconformidades com a Regra de Dependências da Clean Architecture em Python

São José dos Campos, SP

Clevio Orlando de Oliveira Junior

Uma Ferramenta para a Detecção de Inconformidades com a Regra de Dependências da Clean Architecture em Python

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Universidade Federal de São Paulo – UNIFESP

Instituto de Ciência de Tecnologia

Bacharelado em Ciência da Computação

Orientador: Prof. Dr. Fábio Fagundes Silveira

São José dos Campos, SP

11 de Fevereiro de 2022

Clevio Orlando de Oliveira Junior

Uma Ferramenta para a Detecção de Inconformidades com a Regra de Dependências da Clean Architecture em Python

Trabalho de conclusão de curso apresentado ao Instituto de Ciência e Tecnologia – UNIFESP, como parte das atividades para obtenção do título de Bacharel em Ciência da Computação.

Trabalho aprovado em 11 de Fevereiro de 2022:

Prof. Dr. Fábio Fagundes Silveira
Orientador

Prof. Dr. Otavio Augusto Lazzarini Lemos
Membro

Prof. Dr. Tiago Silva da Silva
Membro

São José dos Campos, SP
11 de Fevereiro de 2022

Dedico este trabalho ao meu gato, Aria, que durante este período único de evolução e amadurecimento intelectual, foi a companhia mais pura e sincera que poderia desejar, e tornou meus dias, até os mais difíceis, os mais doces, com seu amor. Que este trabalho eternize a nossa amizade e cooperação, e possamos realizar muitos outros juntos.

Agradecimentos

Agradeço primeiramente ao meu gato, Aria. Sem ele não haveria suporte, motivação e energia para buscar constantemente novo conhecimento. Através de sua mais pura amizade e companhia, sou capaz dentre muitas coisas, de dedicar à minha vida acadêmica de forma sustentável.

Agradeço aos meus pais, Clevio e Kátia, que foram capazes de me proporcionar um ambiente adequado, alimentação balanceada, e condições favoráveis à produção intelectual durante o meu período na universidade. Além disso, foram capazes de, desde a infância, proporcionar cursos de idioma, educação e ferramentas de alta qualidade para o meu desenvolvimento intelectual, sem medir esforços, e sem a ajuda deles, não haveria condições favoráveis para frequentar a faculdade ou escrever este trabalho.

Agradeço ao único irmão que conheci, Bráulio, que já experiente na área de Engenharia de Software, me capacitou a pesquisar por conta própria as minhas dúvidas e buscar soluções, me apresentou diversos paradigmas de programação no meu primeiro ano de faculdade, me apresentou diversas formas de escrever código limpo, e mais importante, me mostrou outros sistemas operacionais e despertou intenso interesse pelos meus próprios computadores. De sua própria forma indireta, me inspirou a escolher que o assunto deste trabalho fosse engenharia de software, no lugar de outras áreas da computação.

Agradeço ao projeto interlinguando, que me engajou no meio acadêmico e me forneceu a oportunidade de desenvolver minhas habilidades de ensinar e escrever durante a faculdade, através do tempo que fui instrutor do idioma japonês no projeto, pois essas habilidades são grande parte das que uso para escrever este trabalho. Também agradeço ao corpo docente da Universidade Federal de São Paulo, pois me forneceram um ambiente técnico favorável para me aprofundar no conhecimento e desenvolvimento científico necessários para a escrita deste trabalho. Agradeço ao meu orientador, Prof. Fábio, por dispor de seu tempo para orientar e auxiliar na síntese do tema deste trabalho, recomendar artigos acadêmicos e dicas de como escrever um TCC, revisar o conteúdo escrito e recomendar diversas possíveis melhorias através de sua extensa experiência acadêmica, na escrita deste trabalho.

Agradeço à todas as pessoas que me deram suporte emocional e psicológico durante todos os anos na faculdade, e agradeço à todas as pessoas que me deram qualquer tipo de suporte durante a criação desse trabalho.

Por fim, agradeço a todos os meus amigos de infância, de escola, e os que conheci na internet, que me proporcionaram memórias inspiradoras as quais me apoio nos momentos que é preciso ter perseverança e continuar realizando um bom trabalho: Victor Prodoscimo, Adrian Castello, Matheus Andrello, Gustavo Stande, Gabriel Negre, Hyun Soo Kim, Pedro

Negrini, Diego Lisboa, Marayane Kálita, Felipe Augusto, e tantos outros. Agradeço a todos que enriqueceram minha jornada na universidade, desde as amizades sinceras e meus alunos no interlinguando, aos pequenos mas valiosos conselhos. Dentre todas as figuras especiais, gostaria de destacar: Matheus Galeote, Leonardo Yuji Yocoyama, Felipe Calderan e Luiz Otávio Passos. Muito obrigado.

"Quando um adulto diz: "O processo vale mais que os resultados"
crianças não são capazes de entender. Porém eu estou do lado dos adultos;
minha vida é construída a partir do meu dia-a-dia, e resultados não passam de subprodutos
(Kita Shinsuke, HAIKYU!!: TO THE TOP Episódio 20, Tradução própria)

Resumo

Devido à necessidade encontrada na Engenharia de Software de criar aplicações com maior coesão e menor acoplamento, Robert C. Martin propôs a *Clean Architecture*. A Clean Architecture e sua obra literária ensinam diversos conceitos para arquitetos de software e desenvolvedores, conceitos estes que buscam o objetivo único de criar aplicações mais estáveis, duráveis e flexíveis, que conseguem distinguir um detalhe de algo crucial para seu funcionamento. Este trabalho canaliza esforços para modelar e adaptar projetos existentes aos preceitos da Clean Architecture através de detecção de bad smells, termo utilizado para descrever trechos de código que sinalizam problemas, e técnicas de refatoração pertinentes à violação dos preceitos da Clean Architecture. É evidente que muito já foi realizado para catalogar e tratar bad smells clássicos, os code smells. No entanto, o mesmo não pode ser dito para architecture smells, termo também derivado de bad smells, mas com um foco em arquiteturas de software. A fim de encontrar e catalogar maneiras eficazes de realizar refatoração, foram realizadas pesquisas sobre as diferentes formas que os bad smells podem se manifestar, pois, tendo esse conhecimento, se torna factível a detecção destes bad smells e pode ser realizado um experimento em diferentes aplicações a fim de propor novas formas de refatorar o design de uma aplicação. Neste trabalho, os esforços foram direcionados à linguagem Python, e com inspiração em trabalhos realizados para a Clean Architecture e outros estilos arquiteturais como o MVC, em outras linguagens de programação, como Java, PHP e C#, foram agrupadas e comparadas diversas ferramentas de análise estática de código a fim de desenvolver uma maneira de utilizá-las em conjunto para estabelecer regras que identifiquem de forma programática a violação da regra de dependências para a Clean Architecture.

Palavras-chaves: Engenharia de Software, Clean Architecture, Bad Smells, Refatoração.

A Tool to Enforce the Clean Architecture's Dependency Rule in Python

Abstract

Clean Architecture was first introduced by Robert C. Martin to address the need for more loosely coupled components and better cohesion. The Clean Architecture focuses on preparing software engineers to write more stable, durable and flexible applications capable of distinguishing between mere details (e.g. what framework it uses) and business logic vital to its functioning. This work directs efforts towards redesigning and readapting existing programs to the Clean Architecture through detecting so-called bad smells - a term used to describe when a piece of code might lead to problems in the future - that break Clean Architecture constraints and finding refactoring techniques to solve these smells efficiently. The literature shows that much has been done to catalogue and solve bad smells related to code, often called code smells. However, the same does not apply to architecture smells - its software architecture counterpart. First, research has been done to find different ways architecture smells manifest so that later, it becomes easier to experiment and compare different refactoring techniques that impact existing projects' design. Even though the chosen programming in this work has been Python, thorough research has been done on previous papers showing similar work written in Java, PHP, or C# and other architectural styles such as the MVC. That allows grouping and comparing static code analysis tools and find a way to reuse them to enforce Clean Architecture's Dependency Rule programmatically.

Key-words: Software Engineering, Clean Architecture, Bad Smells, Refactoring.

Lista de ilustrações

Lista de abreviaturas e siglas

I/O	Input/Output ou Entrada e Saída
SRP	Single Responsibility Principle
SPL	System Product Lines
smell	Bad Smell
pub-sub	Publisher-Subscriber pattern
ADP	Acyclic Dependencies Principle
DIP	Dependency Inversion Principle
JSON	JavaScript Object Notation
HTML	HyperText Markup Language
CI/CD	Continuous Integration & Continuous Delivery
CLI	Command-line Interface
AST	Abstract Syntax Tree
ISP	Interface Segregation Principle
SCP	Separation of Concerns Principle
MVC	Model View Controller
ORM	Object–Relational Mapping

Sumário

1 Introdução

1.1 Contextualização e Motivação

1.1.1 Introdução à técnicas de refatoração

Técnicas de refatoração se mostram constantemente necessárias no mercado de trabalho. Isso ocorre pois, em projetos multidisciplinares, é comum que, devido aos prazos, programadores optem por atalhos, estes resultam em *tradeoffs* que permitem com que as expectativas do cliente sejam atingidas a curto prazo, acumulando-se dívidas técnicas que inflam o custo de manutenção a longo prazo de uma aplicação (GUO et al., 2011).

Durante as últimas décadas, foram desenvolvidas diversas maneiras de se realizar refatoração. Essas formas costumam formar-se primeiramente apontando um problema, comumente chamado de *code smell* (FOWLER, 1999) em contraste com um exemplo de código correto. Exemplos de *code smells* mencionados na literatura incluem código duplicado e classes "grandes" demais (demasiadas funcionalidades primárias).

Dentre os métodos que possibilitam tais detecções, está a análise do grafo de dependências de um programa (FERRANTE; OTTENSTEIN; WARREN, 1987). Esse grafo é direcionado e representa dependência entre elementos de um programa, e pode ser utilizado para, por exemplo, detectar códigos duplicados (HOTTA; HIGO; KUSUMOTO, 2012), o que configura um *code smell* proposto por (FOWLER, 1999).

1.1.2 Motivação

A motivação deste trabalho pode ser relacionada com o trabalho realizado no Projeto Case (BELTRãO; FARZAT; TRAVASSOS, 2020), onde o autor implementa, em seu ambiente de trabalho, mecanismos de refatoração direcionados especificamente para corrigir inconformidades com a Clean Architecture (MARTIN, 2017).

Diversos livros e sites (como o *Refactoring Guru*[1]) propõem técnicas de refatoração de código. No caso deste trabalho, serão sugeridas técnicas de refatoração com foco nas especificações da Clean Architecture, e meios de detectar *bad smells* relacionados à inconformidades desta, de forma programática, através da análise de código-fonte.

No *Projeto Case* (BELTRãO; FARZAT; TRAVASSOS, 2020) são listados parâmetros para categorizar os *plugins* que procuram automatizar a detecção de dívida técnica. O parâmetro *Architecture Layer* especifica que camada da arquitetura uma dada regra pode atuar. Juntamente

[1] https://refactoring.guru/

com a análise entregue pelo *Roslyn* [2] e o grau de severidade estimado, o resultado se mostrou positivo, criando uma rotina de refatoração contínua na equipe.

Este trabalho pode ser considerado uma alternativa do realizado em (VELASCO-ELIZONDO et al., 2017), pois possui essência similar, diferindo-se no modelo arquitetural e linguagem de programação escolhidos.

1.1.3 Linguagem python

Para este trabalho, foi escolhida a linguagem de programação Python, tendo em mente o crescimento da popularidade desta linguagem no meio acadêmico, e a oportunidade mostrada visto que na maioria dos casos a Clean Architecture é estudada em linguagens estaticamente tipadas, e o Python, uma linguagem dinamicamente tipada.

Como pré-requisito do trabalho, é necessário conhecer as ferramentas de análise de código disponíveis para Python. Segundo o site *analysis tools*[3], que é mantido pela comunidade de código aberto e revisado diariamente, existem 82 ferramentas de análise estática de código[4]. No Capítulo 3, apresenta-se uma comparação entre tais ferramentas.

A análise estática é o processo de análise de um programa sem realizar a execução do mesmo (LOURIDAS, 2006), em contraste com a análise dinâmica, que realiza a análise em tempo de execução. A análise estática é um processo computacionalmente barato se comparado com a análise dinâmica. Dentre diversas aplicações, é utilizada para verificar *bad smells* (SZŐKE et al., 2015) (WALKER; DAS; CERNY, 2020) através de regras definidas previamente, procurar *bugs* e vulnerabilidades de segurança (GOSEVA-POPSTOJANOVA; PERHINSCHI, 2015).

Exemplos de aplicações de análise estática são comumente encontrados em ferramentas de desenvolvimento de *software*, como *IDEs* e editores de texto[5,6]. Essas aparecem muitas vezes em forma de *plugins* ou ferramentas de linha de comando. Além disso, existem servidores de linguagens de programação[7,8], implementados utilizando o *Language Server Protocol* (LSP) (FOUNDATION, 2016), que interagem com os editores de texto para fornecer análise estática durante o desenvolvimento de aplicações.

[2] https://github.com/dotnet/roslyn
[3] https://analysis-tools.dev/tag/python
[4] Quantia observada em: 22/07/2021.
[5] https://www.jetbrains.com/help/pycharm/code-inspection.html
[6] https://code.visualstudio.com/docs/python/linting
[7] https://github.com/apple/sourcekit-lsp
[8] https://github.com/python-lsp/python-lsp-server

1.2 Definição do Problema

Por muito tempo, tinha-se que refatorar significava esconder as partes do código que lidavam com a entrada e saída através de sub-rotinas (WHEELER, 1952). Hoje em dia, vê-se que refatorar significa algo diferente. Dessa vez, mais próximo de expor as operações de *I/O* e esconder as operações que lidam com a lógica de negócio (RHODES, 2014).

Isto se relaciona e se mantém na Clean Architecture (RHODES, 2014). O *I/O* também é a camada mais externa, é o que é exposto ao usuário e o que aparece primeiro no código. Tudo o que vem de forma mais interna é lógica de negócio. Um bom jeito de estruturar programas pode ser observado na programação funcional (RHODES, 2014) (HUGHES, 1989), que sempre expõe os dados e as estruturas de dados, e eliminam *side-effects* através do conceito de imutabilidade (HAKONEN et al., 1999).

É evidente que independente de como isso seja exposto, o objetivo permanece similar: o de desacoplar código, para que os programas se tornem flexíveis e possam ser constantemente atualizados e expandidos, se esse for um objetivo, pelo maior prazo possível.

As próximas sub-seções organizam em itens os problemas que este trabalho procura resolver.

1.3 Justificativas

Atualmente, muito se discute sobre os benefícios de manutenção que os estilos arquiteturais como a Clean Architecture ou a Hexagonal Architecture fornecem. Porém, pouco se discute sobre como adequar programas já existentes a tais arquiteturas, e menos ainda sobre como detectar violações a seus preceitos fundamentais.

Este trabalho agrega, por se tratar de uma ferramenta de fácil utilização, produtividade no desenvolvimento de *software*. Ao invés de investigar diversos módulos para ver se algum viola tais preceitos, o produto deste trabalho assume a tarefa de navegar pelo código fonte e apontar tais violações.

Por fim, como visto no Capítulo 3 há uma grande escassez de ferramentas que resolvem *bad smells* em estilos de arquitetura, e isso torna este projeto uma contribuição na área, podendo ser inspiração para surgimento de novas ferramentas.

1.4 Objetivos: Geral e Específicos

1.4.1 Objetivo Geral

O objetivo deste trabalho é o de propor técnicas de refatoração de código para a adequação de códigos existentes aos preceitos da Clean Architecture, através do estudo realizado

com o CALint, uma ferramenta criada para este projeto, cuja responsabilidade é criar um conjunto de regras, baseada em quais ferramentas de análise estática de código forem, que capaz de identificar e alertar sobre os *smells* que violem preceitos da Clean Architecture presentes no código fonte de um sistema desenvolvido em Python.

1.4.2 Objetivos Específicos

- Agregar diferentes técnicas de refatoração de códigos que tenham relação com o contexto de arquitetura, de forma que ilustre conceitos e escolhas que podem auxiliar a encontrar, futuramente, técnicas especificas para a Clean Architecture;
- Agregar diferentes *bad smells* encontrados na literatura, a fim de analisar o que pode ser identificado como "violação dos preceitos da Clean Architecture";
- Estender a funcionalidade de um ou mais analisadores estáticos a fim de detectar e alertar sobre *bad smells* que violam preceitos da Clean Architecture;
- Propor técnicas de refatoração para os *bad smells* encontrados.

1.5 Metodologia

1.5.1 Introdução

Nesse trabalho, o princípio a ser abordado de forma mais ampla será a manutenção da regra de dependências (MARTIN, 2017), e as refatorações mais comuns para a sua adequação.

Foi desenvolvido um utilitário em Python, capaz de utilizar os *plugins* ou argumentos corretos, pertinentes às ferramentas selecionadas, para realizar a detecção dos *bad smells*. Esse programa deve ser acessado através da linha de comando.

1.5.2 Adequação as ferramentas

Para definir quais são as camadas arquiteturais de cada aplicação, o usuário ou o responsável pela refatoração deve indicar, em algum momento, quais são os módulos que dizem respeito a cada camada arquitetural, isto é, quais módulos compõem as entidades, casos de uso, e as outras camadas que descrevem a Clean Architecture.

1.6 Organização do Documento

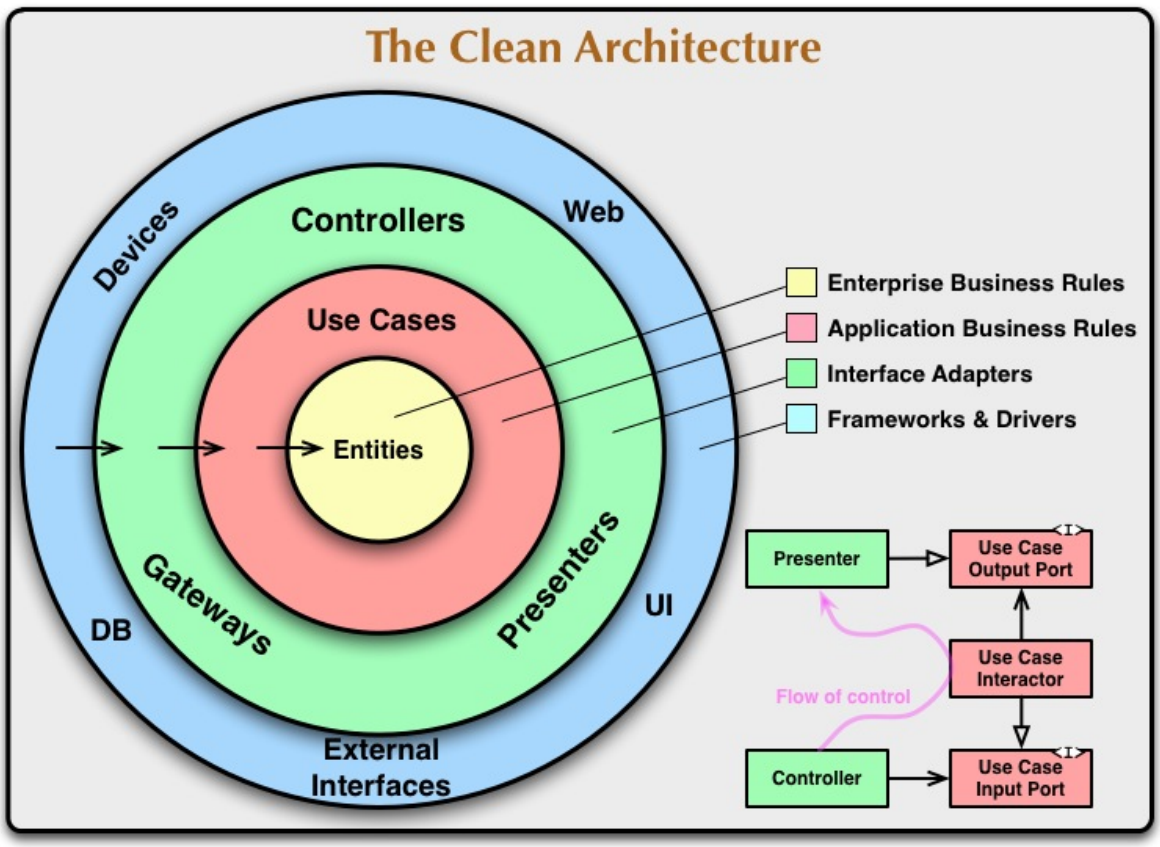
O trabalho se encontra disposto da seguinte maneira:

- No Capítulo 1 se encontra esta introdução, com toda a contextualização e especificações do que é necessário que o leitor conheça para que entenda o que será descrito nos próximos capítulos;
- No Capítulo 2 se encontra a fundamentação teórica que sustenta o trabalho e os estudos realizados neste. Também o assunto que foi aprofundado através da Revisão Bibliográfica;
- No Capítulo 3 se encontra a Revisão Bibliográfica. Este capítulo é destinado para descrever toda a pesquisa feita e base para as escolhas tomadas durante o desenvolvimento. Junto a esta, se encontra um comparativo técnico de ferramentas de análise estática em Python;
- No Capítulo 4 se encontram alguns trabalhos correlatos, e uma explicação do que faz este trabalho ser diferente de outros encontrados na literatura, de forma a provar a autenticidade e a importância deste trabalho;
- No Capítulo 5 está a ferramenta desenvolvida para este trabalho, o CALint, e uma explicação detalhada;
- No Capítulo 6 se encontram os resultados e casos do estudo realizados no próprio CALint como em programas de pequeno porte que anunciam seguir os preceitos da Clean Architecture;
- No Capítulo 7 está a conclusão deste trabalho, assim como funcionalidades que servem de referência para uma continuação do trabalho, e sua importância na academia e na indústria.

2 Fundamentação Teórica

2.1 Clean Architecture

Figura 1 – A Clean Architecture (MARTIN, 2012)



"The way you keep software soft is to leave as many options open as possible, for as long as possible. What are the options that we need to leave open? They are the details that don't matter." (MARTIN, 2017)

A Clean Architecture é uma solução de Engenharia de Software proposta por (MARTIN, 2017). Esta solução é inspirada na *Boundary-Controller-Entity* (ou BCE) Architecture, escrita por (JACOBSON, 1992), e em arquiteturas similares, sendo as de maior prestígio, a Hexagonal Architecture (COCKBURN, 2005) e a Onion Architecture (PALERMO, 2008).

As diversas arquiteturas mencionadas possuem, dentre outros, o princípio e objetivo comum de desacoplar componentes e construir *software* de melhor qualidade e durabilidade. Esses objetivos são atingidos através da separação de camadas e contexto. O autor da obra

menciona, em uma de suas palestras (MARTIN, 2019), que a aplicação na Clean Architecture, de forma análoga ao UNIX (RITCHIE, 1990) que abstrai a tela ou a impressora como o *stdout* ou *standard output*, deve funcionar sem conhecer o dispositivo de entrada e saída. Essa analogia é estendida diversas vezes para justificar que o programa não deve conhecer a plataforma ou o banco de dados que pretende operar sobre. Essa plataforma, portanto, segundo (MARTIN, 2017) deve apenas ser considerado um "detalhe". Em (MARTIN, 2019) o autor menciona o problema de muitas aplicações *rails*[1] serem facilmente reconhecidas, ao ler a estrutura de pastas, por serem aplicações *rails*, enquanto o correto seria, serem reconhecidas pelos problemas que resolvem.

Esses *bad smells*, segundo (MARTIN, 2017), muitas vezes são consequência dos programas misturando esses níveis de abstração. Por isso, ele propõe, em sua arquitetura, que todas as camadas devem ser implementadas de forma unidirecional. Dessa forma, as camadas mais externas interagem com as mais internas, mas não o contrário.

2.1.1 Entities ou Entidades

As entidades são o núcleo da aplicação. Segundo (MARTIN, 2017), é a parte da aplicação que deve funcionar independentemente de existir uma solução computacional para o problema a ser resolvido. Por exemplo, caso o problema seja de criar uma aplicação que trata e valida diversos dados que vem de uma planilha, as entidades seriam todas as regras que classificam e validam os dados, não se importando em como esses dados são fornecidos.

2.1.2 Use Cases ou Casos de uso

Os casos de uso "orquestram" (MARTIN, 2012) as entidades no único sentido que eles são responsáveis por executar as operações que as entidades possuem. Essa ponte existe para garantir que os casos de uso sejam os responsáveis pela lógica de negócio da aplicação.

Mantendo o exemplo da seção anterior, os casos de uso seriam os que recebem os dados da planilha, e executam as operações das entidades de forma que se adéquem à lógica de como os casos de uso imporem o funcionamento desta planilha. Perceba que mesmo o caso de uso sabendo como a planilha funciona, eles não sabem dizer o formato visual ou onde será apresentada, afinal são eles que implementam a lógica de negócio da aplicação, por isso permanecem sem o conhecimento de como apresentar, tratar ou consumir esses dados.

2.1.3 Adapters ou Adaptadores

Os adaptadores são responsáveis por controlar os casos de uso. Eles definem como querem receber os dados externos e como vão retornar esses dados. Apesar de não dependerem de como as partes externas irão utilizá-los, é conveniente possuir adaptadores que transportem

[1] https://rubyonrails.org

os dados de forma útil (MARTIN, 2012), dependendo da requisição, mas que transportem esses dados de forma mais abstrata possível, seguindo a regra de dependências.

Esse fluxo garante que nem mesmo os adaptadores saibam como a *framework*, ou a plataforma, pretende utilizar esses dados e como eles querem utilizar esses dados, dado que as definições que constroem os adaptadores são implementadas conhecendo apenas as camadas mais internas, e estas por sua vez não os conhecem.

Nesta camada se encontram 3 principais adaptadores: *Controllers*, *Presenters* e *Gateways*. Segundo (MARTIN, 2017), existe um fluxo de controle sobre os casos de uso. Esses fluxos de controle fazem uso extensivo, assim, como as outras camadas, do princípio de inversão de dependências (para que os adaptadores dependam dos casos de uso), e esse fluxo de controle é apresentado da seguinte forma (Figura 1):

- O *Controller* depende da porta de entrada do caso de uso;
- Os dados passam pelo caso de uso, e chegam até o *Presenter*;
- O *Presenter* implementa a porta de saída do caso de uso e impõe sua própria forma de apresentar esses dados.

2.1.4 Frameworks e drivers

Os *frameworks* e *drivers* constituem o *I/O* da aplicação. É a parte que deve ser mais facilmente substituível.

Isso inclui: o banco de dados, a aplicação *web*, a plataforma *mobile*, e qualquer outra forma de armazenamento de dados ou interação com o usuário, assim como toda a organização externa de como essa infraestrutura será organizada e que protocolo será o responsável por transportar os dados dos adaptadores.

Note que, apesar desta organização de camadas ser o suficiente em muitos casos, (MARTIN, 2017) também propõe que podem existir mais camadas externas, desde que a regra de dependências seja mantida.

2.2 Bad Smells

> *"Common wisdom suggests that urgent maintenance activities and pressure to deliver features while prioritizing time-to-market over code quality are often the causes of such smells." (TUFANO et al., 2015)*

Code Smells (FOWLER, 1999), *Design Smells* (MARTIN, 2003) ou *Anti Patterns* (BROWN, 1998) são três termos semelhantes, pois descrevem problemas que pertinentes ao processo de desenvolvimento de *software*. O termo mais famoso deles, *Code Smells*, descreve

problemas especificamente relacionados a código. *Design Smells*, descreve problemas estruturais de design de código e engenharia de software, ao passo que *AntiPatterns* é o termo mais amplo, pois alcança problemas de arquitetura de *software*, *design* de *software*, gerenciamento de recursos e projetos.

Apesar das sutis diferenças, as soluções para ambos os três se afunilam na refatoração, e além disso, buscam atingir o mesmo objetivo em engenharia de *software*: aplicações flexíveis (MARTIN, 2009), facilmente modificáveis e desacopladas (KERIEVSKY, 2005), enquanto mantém sua funcionalidade. É importante ressaltar que, apesar de indesejáveis, *code smells* normalmente não causam *bugs* ou alteram o funcionamento do programa (SURYANARAYANA, 2015), normalmente apenas mostram partes sujeitas a menor flexibilidade, maior dificuldade de manutenção, e dependendo da gravidade do *smell*, pode ser que sejam motivos de *bugs* em outras partes do código.

2.2.1 Clean Architecture e Architecture Smells

> "This rule also pertains to source files, components, and modules. Good software design requires that we separate concepts at different levels and place them in different containers. Sometimes these containers are base classes or derivatives and sometimes they are source files, modules, or components. Whatever the case may be, the separation needs to be complete. We don't want lower and higher level concepts mixed together."(MARTIN, 2009)

Como sugere (TUFANO et al., 2015), *bad smells* e dívida técnica estão fortemente associados. Parte da solução da dívida técnica envolve controlá-los através da refatoração. Para a Clean Architecture, resolver significa adequar códigos que violem os padrões da arquitetura e seus princípios.

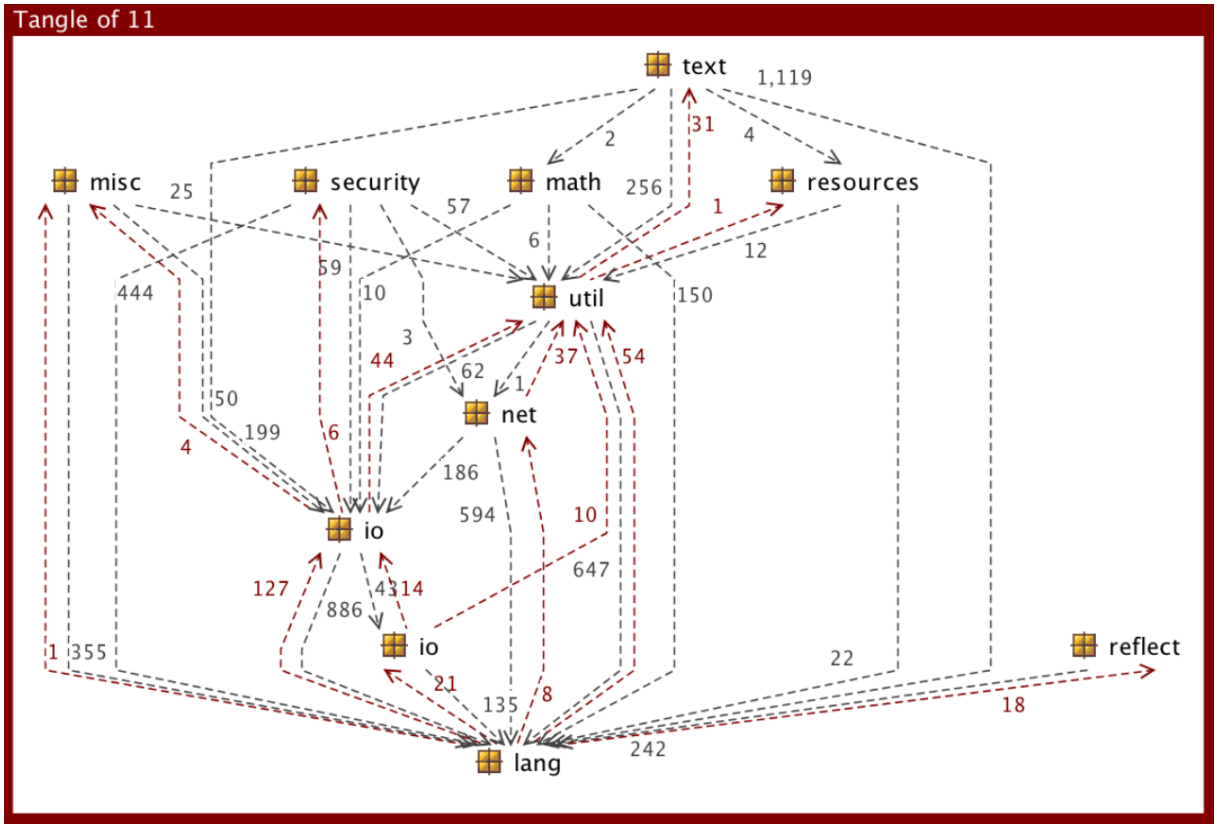
O conhecimento sobre *Architecture Smells* ainda está por atingir a "maturidade" (SAMARTHYAM; SURYANARAYANA; SHARMA, 2016) ideal. Porém, é possível identificar tais *smells* (MALIK et al., 2020) (GARCIA et al., 2009). Pode-se observar que muitos destes possuem similaridade com possíveis causas de quebras dos preceitos da Clean Architecture.

Por exemplo, o Cyclic Dependency Smell (SURYANARAYANA, 2015), é um smell que aponta código tal que dois módulos, classes, ou arquivos apontem dependências entre si, acoplando-os. Em (SAMARTHYAM; SURYANARAYANA; SHARMA, 2016) é disposta a Figura 2, criada utilizando a ferramenta "Structure 101"[2]. Ter uma dependências cíclica, pode ser considerado um indício de quebra da Clean Architecture, pois, se essa dependência atravessa uma camada das camadas definidas em (MARTIN, 2017), há quebra da regra de dependências, e do ADP.

Em Python, esse problema costuma ser raro e requer maior engenharia, pois o interprete da linguagem para de funcionar quando chega em um ciclo de *imports* (VUKASINOVIC, 2020).

[2] https://structure101.com

Figura 2 – Cyclic Dependency Smell, identificado na JDK (SURYANARAYANA, 2015)



É evidente que o trabalho possui um grande escopo, e o problema a ser enfrentado é maior do que algo que dá pra resolver apenas através de programação. É necessário reduzir a escassez relacionada ao assunto de refatoração de arquiteturas, estudar e selecionar *smells* que digam respeito à Clean Architecture, dentre estes, selecionar os que podem ser observados em projetos feitos em Python, e assim, implementar extensões ou *plugins* de ferramentas já existentes que possam alertar sobre estes *smells*.

3 Revisão Bibliográfica

3.1 Introdução

A pesquisa bibliográfica realizada para escrever esse projeto foi dividida da seguinte forma:

1. Entender os diferentes tipos de *bad smells* e como eles afetam a arquitetura de uma aplicação;
2. Procurar possíveis soluções na literatura para esses *smells*;
3. Procurar diferentes técnicas de refatoração para estes *smells*;
4. Descobrir ferramentas comumente utilizadas para detectá-los.

3.2 Architecture Smells

Em (GARCIA et al., 2009) é observado que *code smells* clássicos apenas se baseiam em problemas de implementação. Portanto, se faz necessário encontrar *architecture smells*, pois descrevem problemas na arquitetura do *software*.

Esse conceito foi inicialmente descrito em (LIPPERT, 2006). Os *smells* em (GARCIA et al., 2009) podem ser observados tanto na etapa de planejamento quanto na implementação. *Architecture smells*, diferente de outros *smells* são definidos de forma mais ampla, e costumam estar mais fortemente relacionados com o *design* da aplicação, ao invés do código (GARCIA et al., 2009). Após pesquisar e analisar o código fonte presente em sistemas industriais, (GARCIA et al., 2009) encontra 4 *architecture smells*. Esses *smells* foram apresentados com uma descrição, uma figura apresentando a ligação dos artefatos, os impactos na qualidade e seus *tradeoffs*. Também foi incluído um exemplo em sistemas industriais. Os *smells* definidos pelo autor foram: *Connector Envy*, *Scattered Parasitic Functionality*, *Ambiguous Interfaces*, *Extraneous Adjacent Connector*. Em (ANDRADE; ALMEIDA; CRNKOVIC, 2014) é adicionado um quinto *architecture smell*: *Feature Concentration*. Este por sua vez é um *smell* encontrado em *System Product Lines* (SPL), um paradigma que visa reutilizar componentes em diferentes aplicações (ANDRADE; ALMEIDA; CRNKOVIC, 2014). Segue uma resumida descrição dos *smells*.

- *Connector Envy*: esse *smell* se caracteriza por conectores que possuem funcionalidade extras além de serem apenas conectores. Isso é prejudicial, uma vez que torna o programa

mais inflexível e difícil de testar (GARCIA et al., 2009). Esse smell pode ser relacionado com uma quebra do SRP (MARTIN, 2017).

- *Scattered Parasitic Functionality*: este *smell* viola o *Separation of Concerns Principle* (SCP)(WIN et al., 2002) pois, segundo (GARCIA et al., 2009) ele é encontrado quando um componente desmembra a sua responsabilidade em outros componentes, e esses outros componentes, são responsáveis por propósitos múltiplos que não possuem relação com o propósito inicial.

- *Ambiguous Interfaces*: Costumam ocorrer em interfaces que fazem uso intensivo de *Observers* ou *publisher-subscriber patterns*, segundo (GARCIA et al., 2009), pois esse *smell* é caracterizado por uma função que passa por diversos condicionais que localizarão a função a ser executada. Quando a interface é muito abrangente, é possível que a análise estática das dependências entre os componentes seja prejudicada (GARCIA et al., 2009). Essas interfaces ambíguas são facilmente encontradas em classes que possuem apenas um método responsável por prover múltiplas funcionalidades e os tipos aceitos são muito genéricos (GARCIA et al., 2009). Uma interface que cuida de menos coisas, e é mais específica, facilita a manutenção e o entendimento da arquitetura (GARCIA et al., 2009), pois fica mais claro a sua função e quais são os possíveis *subscribers*, caso utilize um *Observer*. Além disso, é possível ver que esse *smell* também viola o *Interface Segregation Principle* (ISP) (MARTIN, 2000) por gerar uma grande interface geral no lugar de pequenas interfaces mais específicas.

- *Extraneous Adjacent Connector*: Segundo (GARCIA et al., 2009), é um *smell* que acontece quando dois ou mais componentes se comunicam por mais de uma única maneira. Em (GARCIA et al., 2009) o exemplo dita dois componentes que se comunicam através de *pub-sub* e também de chamada de métodos. Mesmo que cada tipo de comunicação pode ter suas próprias vantagens e desvantagens, utilizar mais de um conector ao mesmo tempo ligando dois componentes carrega o risco de que interfiram na funcionalidade desejada, executem múltiplas vezes o mesmo comportamento, ou se cancelem (GARCIA et al., 2009). Além disso, isso prejudica o entendimento da arquitetura, pois adiciona complexidade no fluxo de comunicação.

- *Feature Concentration*: Este *smell* é específico de SPLs, e é descrito em (ANDRADE; ALMEIDA; CRNKOVIC, 2014). Sua principal característica fica evidente ao concentrar muitas *features* em um único SPL. Em (ANDRADE; ALMEIDA; CRNKOVIC, 2014) é exemplificado que esse *smell* representa o lado oposto do *Scattered Parasitic Functionality*, visto que ao invés de espalhar funcionalidades, ele concentra funcionalidades. A maior consequência negativa deste *smell* é ter um componente mais robusto do que deve ser, e assim, violar o CRP (MARTIN, 2017), pois em um componente robusto com

muitas funcionalidades ortogonais, é comum que existam componentes que poderiam ser reutilizados como pacotes separados.

3.3 Dependências cíclicas e técnicas de refatoração

No Capítulo 14 de (MARTIN, 2017), é descrito o *Acyclic Dependencies Principle* (ADP). Violar este princípio pode ser observado como um *architecture smell* (MALIK et al., 2020)(RIZZI; FONTANA; ROVEDA, 2018)(VELASCO-ELIZONDO et al., 2017)(MACCORMACK; STURTEVANT, 2016)(FONTANA et al., 2017).

No trabalho em (RIZZI; FONTANA; ROVEDA, 2018) é desenvolvido um meio matemático de remover dependências cíclicas utilizando o conceito de grafos. Esse método foi implementado utilizando a ferramenta *Arcan* (FONTANA et al., 2017). Nesse método são feitas múltiplas refatorações e (RIZZI; FONTANA; ROVEDA, 2018) afirma que possui os melhores resultados em menos passos. Em sua conclusão (RIZZI; FONTANA; ROVEDA, 2018) explica que para remover os *smells* a nível de classe e pacote, foram utilizadas três técnicas: *Extract class*, *Move method*, *Move class* e a *design pattern Singleton*. Observe no entanto que, a refatoração de métodos não foi implementada, e as opções de sugestões de refatoração são geradas pelo *Arcan* (RIZZI; FONTANA; ROVEDA, 2018).

Extract class: Encontrada no Capítulo 7 de (FOWLER, 1999), a técnica de refatoração *extract class* é o inverso da técnica *inline class*, nessa técnica, apenas se extraem campos e métodos de um classe, visto que ela "está fazendo o trabalho de duas" (FOWLER, 1999).

Move method: Segundo (FOWLER, 1999), o *Move method* é uma técnica que move o método de uma classe para outra pois esse método é frequentemente utilizado pela outra classe.

Move class: Análogo ao *Move method*, o *Move class* move uma classe de um pacote para outro.

Singleton: Em (GAMMA, 1995) é descrito que o *Singleton* é uma *design pattern* responsável por garantir que exista apenas uma única instância de uma classe na aplicação, pois essa classe é apenas acessível através de um *getter*, e possui um construtor privado.

O *Dependency Inversion Principle* (DIP) (MARTIN, 2003)[1] diz que módulos de alto e baixo nível devem depender apenas de abstrações, e abstrações não devem depender de detalhes, os detalhes devem depender de abstrações. Uma das possíveis implementações é através de uma injeção de dependências (PRASANNA, 2009). Esta *design pattern* permite uma classe qualquer, que implementa uma dada interface, ser utilizada dentro de uma outra classe. Como o método que é chamado está na interface, e todas as classes que a implementam o possuem, a injeção de dependências permite que a abstração não conheça os detalhes, já que os detalhes são

[1] Artigo disponível em: https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/granularity.pdf

implementados em uma classe que implementa a abstração, e depois utilizados em uma classe que, analogamente, conhece apenas a abstração.

Além disso, o experimento realizado em (MACCORMACK; STURTEVANT, 2016) conclui que o acoplamento está altamente ligado a direção das dependências de um programa. Como mencionado diversas vezes em (MARTIN, 2017), as dependências do programa não devem ser bidirecionais.

3.4 Architecture Smells e MVC

Segundo (VELASCO-ELIZONDO et al., 2017), apesar de muitas pessoas saberem do MVC, muitos ainda não são capazes de implementá-lo corretamente. Segundo (VELASCO-ELIZONDO et al., 2017) o MVC tem as seguintes limitações:

1. O *Model* e o *View* não devem lidar diretamente com o processamento de *requests* do usuário. Esta é uma responsabilidade do *Controller*.

2. O *Model* e o *Controller* não devem lidar diretamente com a apresentação dos dados, por exemplo, ele não deve conter código *HTML* ou *JSON*. Esta é uma responsabilidade do *View*.

3. O *View* e o *Controller* não devem lidar com chamadas ao banco de dados. Esta é uma responsabilidade do *Model*.

A partir destes três itens, o autor formulou 6 *bad smells* e utilizou uma aplicação chamada *PHP_CodeSniffer* para criar as regras acima. Sua experimentação foi bem sucedida e o *smell* que ocorreu mais vezes foi o "Controller realiza operações ou manipula dados que pertencem ao Model" (VELASCO-ELIZONDO et al., 2017).

O trabalho de (VELASCO-ELIZONDO et al., 2017) solidifica a ideia de definir *bad smells* e analisar código para encontrar esses *bad smells* para facilitar nas técnicas de refatoração de um modelo arquitetural. De forma análoga, também é possível fazê-lo para qualquer outro modelo arquitetural, como a Clean Architecture.

3.4.1 Aplicação em Clean Architecture

Em (BUI, 2017) o autor menciona diversas formas de aplicar conceitos de Clean Architecture em aplicações *Android*. Isso mostra que o interesse por refatorar código de outras arquiteturas para a Clean Architecture também já existe na literatura. Segundo (BUI, 2017) as aplicações *Android* costumam utilizar o padrão MVC, incluindo a aplicação que foi o objeto de estudo principal do trabalho, *Sunshine*, uma aplicação de consulta do clima.

3.5 Comparativo Técnico

Como visto anteriormente neste trabalho, as ferramentas *Arcan* e *Structure101* são comumente utilizadas para analisar e auxiliar na refatoração da arquitetura de um programa. Porém, essas ferramentas não são facilmente acessíveis em Python.

Nessa seção, são listadas as ferramentas de análise de código que podem auxiliar na detecção dos *smells* selecionados acima, mesmo que através de *plugins*.

3.5.1 Linters

Os *linters* são ferramentas gerais de análise de código, capazes de encontrar *bugs* e sinalizar construções ruins (JOHNSON, 1977). Um exemplo de *linter* para a linguagem Python é o Pylint[2]. O Pylint permite a criação de regras, ou *checkers*, customizados.

Esses *checkers* podem ser criados através de *tokens* gerados, análogo ao *PHP_CodeSniffer* (VELASCO-ELIZONDO et al., 2017), através do arquivo analisado puro, ou através de uma AST pelo *linter* através do *astroid*[3].

[2] https://pylint.org
[3] https://github.com/PyCQA/astroid

```
import astroid
from astroid import nodes

from pylint.checkers import BaseChecker
from pylint.interfaces import IAstroidChecker
from pylint.lint import PyLinter

class UniqueReturnChecker(BaseChecker):
    __implements__ = IAstroidChecker

    name = 'unique-returns'
    priority = -1
    msgs = {
        'W0001': (
            'Returns a non-unique constant.',
            'non-unique-returns',
            'All constants returned in a function should be unique.'
        ),
    }
    options = (
        (
            'ignore-ints',
            {
                'default': False, 'type': 'yn', 'metavar' : '<y or n>',
                'help': 'Allow returning non-unique integers',
            }
        ),
    )
```

Exemplo de como criar um checker customizado (LOGILAB; CONTRIBUTORS, 2022).

Outro exemplo de *linter* é o *Import Linter*[4]. Essa ferramenta consegue definir *constraints* para os *imports*. Isso é útil pois torna capaz definir por parte do programar as camadas arquiteturais mencionadas em (BELTRãO; FARZAT; TRAVASSOS, 2020). Essa ferramenta é capaz de estabelecer contratos (Figura 3), impedindo que casos de uso importem adaptadores, por exemplo.

Figura 3 – Exemplo de contrato de import para o modelo arquitetural MVC (IMANKULOV, 2021)

```
=============
Import Linter
=============

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

Analyzed 3 files, 2 dependencies.
---------------------------------

Models don't import views BROKEN

Contracts: 0 kept, 1 broken.


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

Models don't import views
-------------------------

myproject.views is not allowed to import myproject.models:

-   myproject.views -> myproject.models (l.1)
```

[4] https://github.com/seddonym/import-linter/

3.5.2 Formatadores de código

Formatadores de código verificam se o código está se adequando à uma *guideline* de estilo de código, como o PEP8[5]. Exemplos de tais ferramentas são: o *pycodestyle*[6] e o *black*[7]. Além disso, o *autopep8*[8] utiliza o *pycodestyle* para formatar código automaticamente, e o *black* é capaz de formatar automaticamente por padrão.

3.5.3 Servidores de linguagem

Servidores de linguagem tem como objetivo integrar diretamente com editores de texto, e apontam erros durante o momento que o código está sendo escrito. Essas ferramentas são chamadas de *Language Servers*, e exemplos de ferramentas assim são: *jedi*[9] e o *pylance*[10].

3.5.4 Checagem de tipos em Python

Python é uma linguagem "tipada" dinamicamente, por isso, muitos desenvolvedores acabam encontrando defeitos (KHAN et al., 2021) que não são mencionados pelo interprete da linguagem antes da execução. Uma solução para isso envolve os *static type checkers* ou checadores de tipo estático.

O *pyright*[11] é implementado no *pylance*, apesar de que é possível utilizá-lo sem o *pylance*, possui compatibilidade com múltiplos editores de texto e também funciona através de uma CLI. Ambos *pyright* e *mypy*[12] checam as *Type Hints*[13] ou sugestões de tipos, módulos adicionados no Python versão 3.5[14], e apontam erros quando essas sugestões de tipo são violadas.

Adicionalmente, a combinação do *pylance* e do *pyright* fornece inferência de tipos (OSTROWSKI, 2020), que é quando um analisador de código tenta inferir o tipo de uma dada variável.

3.5.5 Ferramentas que atuam durante o CI/CD

Algumas ferramentas podem atuar durante o *CI/CD*, como o *mega-linter*[15], *semgrep*[16], e o *SonarQube*[17], utilizado em (BELTRãO; FARZAT; TRAVASSOS, 2020).

[5] https://www.python.org/dev/peps/pep-0008/
[6] https://github.com/PyCQA/pycodestyle
[7] https://black.readthedocs.io/en/stable/
[8] https://pypi.org/project/autopep8/
[9] https://github.com/davidhalter/jedi
[10] https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance
[11] https://github.com/microsoft/pyright
[12] http://mypy-lang.org/
[13] https://docs.python.org/3/library/typing.html
[14] https://docs.python.org/3/whatsnew/3.5.html
[15] https://nvuillam.github.io/mega-linter/
[16] https://semgrep.dev
[17] http://sonarqube.org

O *mega-linter* é uma ferramenta capaz de analisar código, formatar código automaticamente e encontrar *bugs*, com suporte para dezenas de linguagens de programação, durante o *CI/CD*. Isto é feito com a ajuda de ferramentas e *linters* que já existem no mercado para cada linguagem.

O *semgrep* e o *SonarQube* são semelhantes, porém o *semgrep* adota um modelo *freemium* com uma versão paga estendida, e é capaz de solucionar problemas automaticamente[18]. Ambas ferramentas possuem um foco em realizar inspeção contínua e análise de código para encontrar *bugs*, *smells*, e vulnerabilidades de segurança.

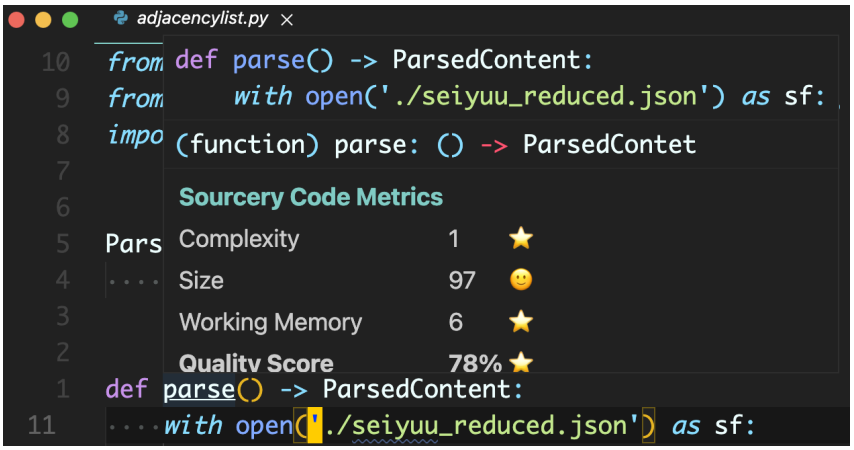
3.5.6 Ferramentas que usam inteligência artificial

Algumas ferramentas de código utilizam algoritmos de inteligência artificial para gerar sugestões de autocompletar. Exemplos destas são o *Kite*[19] e o *GitHub Copilot*[20], treinado no *OpenAI Codex*. Essas ferramentas costumam treinar a partir de uma extensiva base de código (CHEN et al., 2021) para replicar sugestões de trechos de código que, baseado no projeto atual, podem auxiliar na produtividade do programador.

Além disso, em (MIR et al., 2021) é proposto um método de inferência de tipos utilizando aprendizado de máquina, mais abrangente, preciso e performático que métodos de inferência através de análise estática.

3.5.7 Refatoradores de código

Figura 4 – *Sourcery* - métricas de código



Para a linguagem Python, existe uma ferramenta comercial e proprietária chamada *Sourcery*[21]. Esta ferramenta realiza refatoração de código Python automatizada, além de aplicar di-

[18] https://semgrep.dev/docs/faq/#how-is-semgrep-different-than-sonarqube
[19] https://www.kite.com
[20] https://copilot.github.com
[21] https://sourcery.ai

ferentes métricas de qualidade de código. Uma de suas vantagens é que além de funcionar no *PyCharm* e no *Visual Studio Code*, também é possível ativar o *Sourcery* em projetos públicos no *GitHub*.

Em sua versão profissional[22], destinada a engenheiros que desejam refatorar projetos inteiros automaticamente, é capaz de realizar análise de múltiplos arquivos e extrair métodos, realizar refatorações avançadas e detectar código duplicado.

Figura 5 – *Sourcery* - indicação de que uma refatoração pode ser aplicada

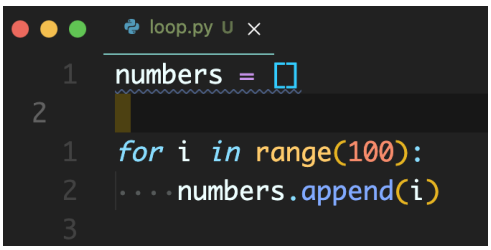
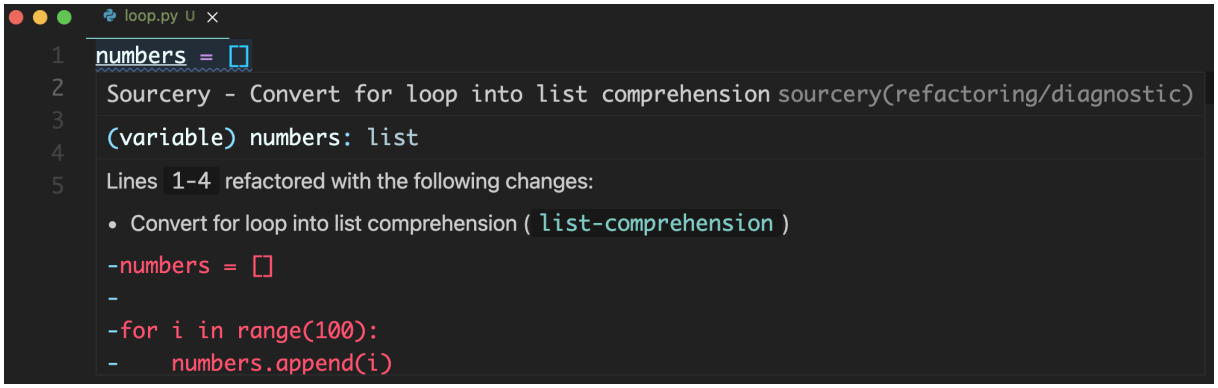
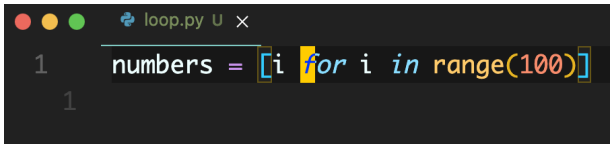


Figura 6 – *Sourcery* - descrição da sugestão de refatoração



O exemplo de métrica de código se encontra na Figura 4, que avalia uma função por sua complexidade, tamanho, e memória. Nas figuras 5, 6, e 7 se está o processo antes da refatoração de um *loop*, a descrição que o *Sourcery* fornece, e o resultado da refatoração automatizada.

Figura 7 – *Sourcery* - após transformar o *loop* em um *list comprehension*



[22] https://sourcery.ai/pro/

3.6 Notas finais

É notável que apesar de existirem diversas ferramentas para análise de *code smells*, existem uma escassez de ferramentas para detecção de *architecture smells* (FONTANA et al., 2017). Sendo as ferramentas encontradas que detectam estes *architecture smells* em sua maioria, relacionadas a dependência entre módulos.

4 Trabalhos Relacionados

O trabalho descrito em (GARCIA et al., 2009) descreve o que são *architecture smells*, e os compara com outros *bad smells*, e encontra, através de trabalhos realizados pessoalmente em diversos sistemas, novos *architectures smells*, os descreve, citando exemplos em sistemas industriais. Estes *smells* são descritos no trabalho, com diagramas, possíveis causas e *tradeoffs*.

Esse trabalho de conclusão de curso também realiza esforços para catalogar *architecture smells*, porém, diferente do trabalho em (GARCIA et al., 2009), esse trabalho reutiliza e aprofunda apenas nos *architecture smells* encontrados que se relacionam com a Clean Architecture, e procura propor explicações e soluções para que os mesmos sejam refatorados e o programa fique com a arquitetura mais próxima da proposta por (MARTIN, 2017). Além de propor uma forma programática de detecção destes.

O trabalho descrito em (BUI, 2017) também é correlato a este trabalho, pois o autor propõe refatorações para a Clean Architecture, porém em um contexto diferente. Primeiro, (BUI, 2017) explica diversas *design patterns* e modelos arquiteturais, como também menciona princípios como a regra de dependências, mas não possui um foco em *bad smells* e técnicas de refatoração, como este trabalho. Adicionalmente, em (BUI, 2017) são mencionados diversas boas práticas assim como seus motivos dentro do contexto de aplicações *Android* e *reactive programming*.

O trabalho de (VELASCO-ELIZONDO et al., 2017) possui o objetivo comum de propor *smells* e técnicas de refatoração para um padrão ou estilo arquitetural. Porém, este trabalho possui um escopo menos bem preenchido, já que a Clean Architecture possui mais camadas e regras do que o MVC. Além disso, esse trabalho foca, primariamente, em *smells* já resolvidos em outros trabalhos, e em formas eficientes de replicá-los na arquitetura proposta por (MARTIN, 2017).

O trabalho de (RIZZI; FONTANA; ROVEDA, 2018) propõe métodos de refatoração para *architecture smells*, mais especificamente, o *Cyclic Dependency Smell*. O trabalho também estende a função de uma aplicação já existente, o *Arcan* (FONTANA et al., 2017), e realiza um experimento em programas grandes, de seu próprio método para remoção desses *smells* utilizando grafos.

Esse trabalho se relaciona, principalmente, com a parte prática deste trabalho. O que torna este trabalho diferente do em (RIZZI; FONTANA; ROVEDA, 2018) é: foco na linguagem Python e nas ferramentas já existentes. A parte prática deste trabalho foca na detecção dos *bad smells* relacionados a possíveis violações da Clean Architecture, enquanto o (RIZZI; FONTANA; ROVEDA, 2018) foca em propor uma solução prática para a remoção automática do *bad smell* mencionado.

O trabalho de (BELTRãO; FARZAT; TRAVASSOS, 2020) descreve o trabalho realizado em um projeto chamado "*Projeto Case*" a fim de reduzir a dívida técnica, que é uma metáfora para programas nos quais os engenheiros terminam o código mais rapidamente em detrimento da qualidade, completude ou segurança (MACCORMACK; STURTEVANT, 2016) (BROWN et al., 2010).

O trabalho realizado implementa regras de detecção de dívida técnica através do *SonarQube* [1], uma ferramenta de inspeção contínua de qualidade de código através de análise estática. Dentre as 52 regras adicionadas ao *plugin* utilizado no projeto, existem regras definidas pelo cliente e regras que diziam respeito a Clean Architecture.

Este trabalho, de forma similar, explora e define regras capazes de detectar práticas inadequadas de programação e de modelagem que ferem alguns preceitos da Clean Architecture, preceitos esses relacionados a *smells* de código, arquitetura, e design. Este trabalho se difere pois, diferente do realizado no *Projeto Case*, o objetivo é centrado em analisar e justificar esses *bad smells*, e a linguagem de programação é o Python, ao invés do C#.

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

5 A Ferramenta CALint

5.1 Introdução

O CALint é uma ferramenta desenvolvida neste trabalho cuja funcionalidade é verificar se um dado projeto em Python está em conformidade com as regras de dependências da Clean Architecture e, caso contrário, apontar os locais em que são verificadas as não conformidades.

Os objetivos do desenvolvimento dessa ferramenta são:

- Utilizar o programa como um objeto de estudo, apontando durante o desenvolvimento as vezes em que a regra de dependências foi infringida, em um processo análogo ao *bootstrapping* (Capítulo 6);
- Facilitar a configuração de outras ferramentas existentes no mercado a fim de estabelecer regras moldadas para a Clean Architecture facilmente, e acelerar o processo de desenvolvimento e refatoração de programas existentes para os preceitos da Clean Architecture, mais especificamente, a regra de dependências;
- Iniciar um projeto de desenvolvimento onde a checagem para regra de dependências se torna contínua e simplificada, através de integrações com ferramentas utilizadas no mercado;
- Criar um exemplo de programa que segue os preceitos da Clean Architecture diferente dos comumente encontrados, que focam primariamente em operações de leitura e armazenamento de dados.

5.2 Implementação

5.2.1 Ferramentas utilizadas

5.2.1.1 Poetry

O *Poetry*[1] é um gerenciador de pacotes para Python, que facilita a instalação de dependências assim como garante que as versões estejam corretas, e também agrega *scripts* para *deploy* e publicação de bibliotecas. O gerenciador de pacotes também cria ambientes virtuais para preparar versões específicas do intérprete do Python.

[1] https://python-poetry.org/

5.2.1.2 Import Linter

O *Import Linter*, discutido no Capítulo 3, foi a ferramenta principal utilizada no projeto. Ao analisar o código da ferramenta, foi realizado uma adaptação das classes e funcionalidades no contexto do projeto, transformando a ferramenta, originalmente projetada para a linha de comando, em uma biblioteca.

5.2.2 Organização do código e arquitetura

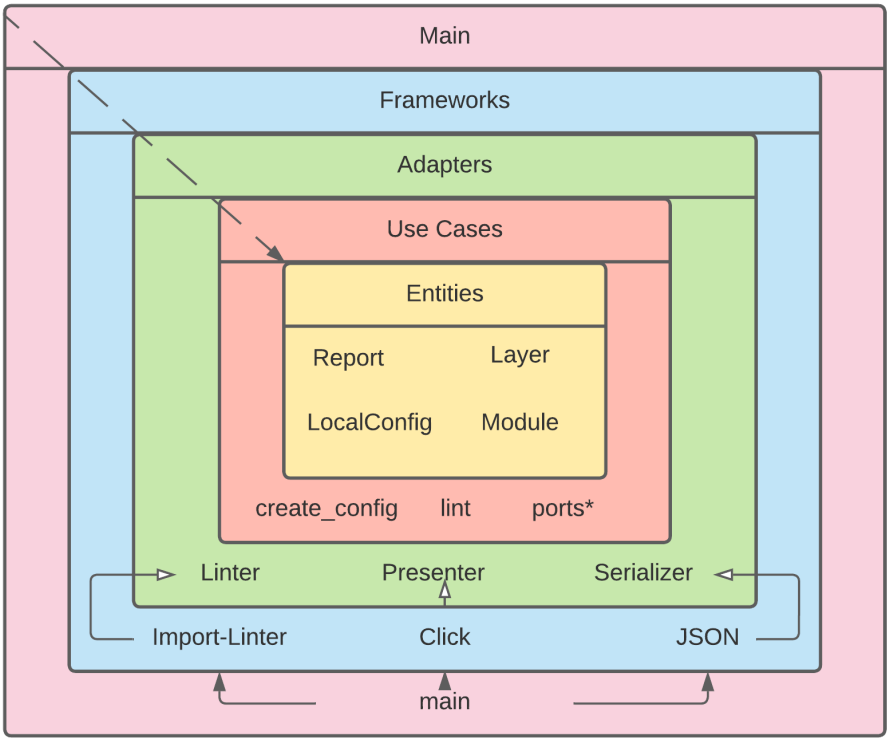
```
calint
├── adapters
│   ├── presenter.py
│   └── ...
├── entities
│   ├── layer.py
│   └── ...
├── frameworks
│   ├── json_serializer.py
│   └── ...
├── use_cases
│   ├── ports
│   │   ├── create_config.py
│   │   └── ...
│   ├── lint.py
│   └── ...
├── .calintrc.json
└── main.py
.gitignore
poetry.lock
prproject.toml
```

Como sugerido por Martin em sua palestra (MARTIN, 2019), a estrutura de pastas deve revelar as funcionalidades do programa, e não ser estritamente orientada a uma *framework* ou a mecanismos de *I/O*. Este projeto utilizou desta premissa para sua organização, as pastas descrevem a localização de cada uma de suas camadas arquiteturais.

O CALint possui 5 camadas arquiteturais: main, frameworks, adapters, use_cases, e entities (Figura 8). Há uma camada extra, o Main. É a porta de entrada para a aplicação. No Main está a função que chama os casos de uso em ordem, enviando parâmetros e injetando dependências em cada um. Primeiro o caso de uso que configura as entidades create_config, em seguida, o caso de uso que cria os resultados, lint.

As *frameworks* são as ferramentas utilizadas no projeto e os serializadores. No CALint

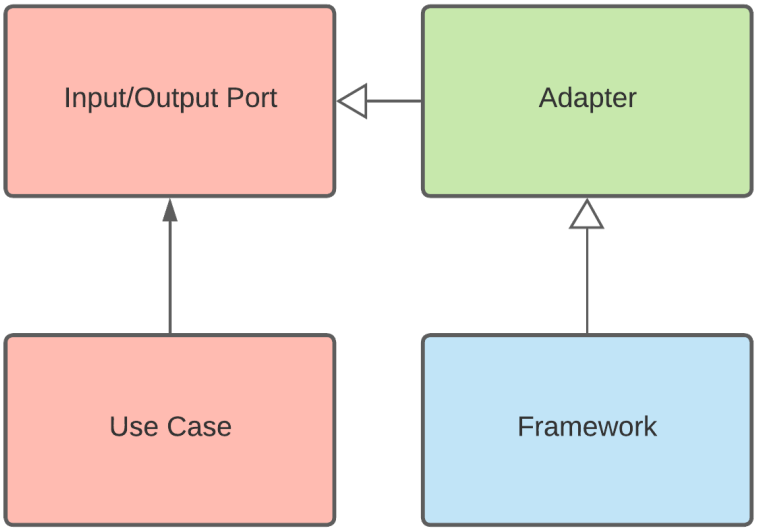
Figura 8 – Arquitetura de dependências do CALint



como o serializador de *json* do Python, uma ferramenta para imprimir texto formatado no terminal, e o *Import Linter*. Esses *frameworks* seguem as especificações criadas pelos adaptadores, que por sua vez, possuem as portas de entradas para os casos de uso. Nesse sentido, os adaptadores controlam os casos de uso, porque os *frameworks* implementam as especificações criadas pelos adaptadores. As *frameworks* utilizadas são as seguintes:

- Click (click_printer.py), foi utilizada, após inspiração do *Import Linter* ao analisar o código fonte para utilizar como biblioteca, com a finalidade de formatar e colorir o *output* da linha de comando.
- Import Linter (importlinter_linter.py), foi utilizado como a principal *framework* no projeto, com o fim de analisar as dependências em camadas. Foi utilizado de forma a configurar um contrato do tipo *"layers"*, já disponível, com as camadas da Clean Architecture na ordem configurada pelo usuário.
- *json* (json_serializer.py), apesar de ser uma biblioteca padrão fornecida pelo Python, é importante que o programador tenha a liberdade de alterar e escolher alternativas de serializadores de dados no projeto, afinal, serializar é um processo de *I/O*. Isso abre portas por exemplo, para, caso exista um requisito de usar outro serializador (e.g. um serializa-

Figura 9 – Fluxo de dependências do caso de uso até o framework



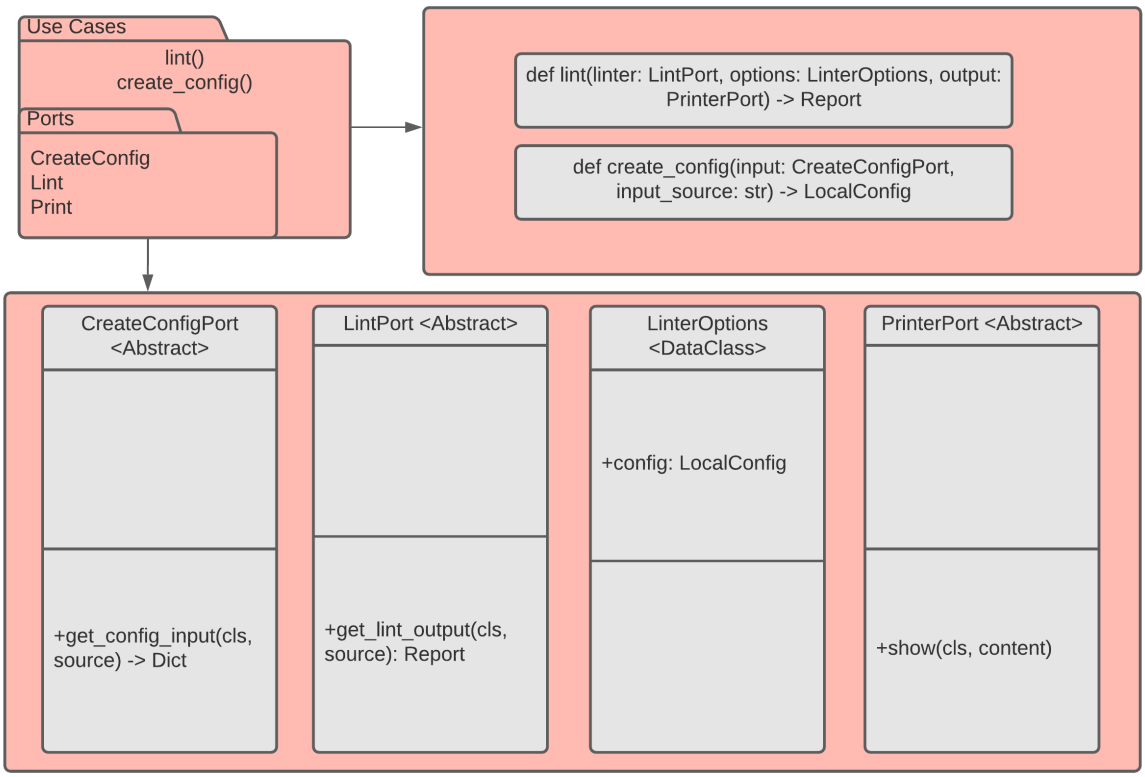
dor para *toml*, biblioteca alternativa de *json*), poder ser trocado facilmente, por conta da injeção de dependências (RAZINA; JANZEN, 2007).

Os *adapters* são os adaptadores do projeto, tanto de entrada quanto de saída. Um adaptador pode ter portas para um ou mais casos de uso, de entrada ou de saída. Mas eles sabem apenas sobre os casos de uso, dependendo que as *frameworks* os implementem para estender sua funcionalidade. No caso, eles fazem as ligações entre as bibliotecas e os casos de uso através da implementação das portas do caso de uso (Figura 9). Os *adapters* são:

- LinterAdapter, que implementa a LintPort. Este *adapter* implementa o método de saída do *linter* chamando um método abstrato, posteriormente implementado por uma das classes que utilizam *frameworks*, chamado lint, que recebe uma configuração do tipo LocalConfig e devolve um Report.
- PresenterAdapter, que implementa a PrinterPort. Este *adapter* implementa a chamada ao present passando o conteúdo do Report, para que o *framework* responsável se encarregue de apresentar o conteúdo do Report.
- SerializerAdapter, que implementa o CreateConfigPort. Este *adapter* implementa a chamada a chamada ao deserialize passando os dados do tipo *string*, após acessar a função read do parâmetro *source* para ler o conteúdo da fonte. Além disso, o SerializerAdapter também disponibiliza a assinatura do serialize, que, de forma oposta ao deserialize, recebe um dicionário e retorna uma *string*.

Os casos de uso coordenam a lógica de negócio da aplicação, populam e regulam os dados das entidades conforme a necessidade da aplicação. Eles não sabem dos adaptadores, por serem os que definem as portas de entrada e saída (classes abstratas). Dessa forma, adaptadores que implementam as mesmas portas podem ser inseridos e removidos conforme a necessidade, sem conhecimento dos casos de uso. Existem 3 portas:

Figura 10 – Casos de uso e portas



- LintPort: Responsável por definir a assinatura do *output* após a execução de um *linter*. Define que a configuração deve ser do tipo LocalConfig e o retorno deve ser uma entidade, o Report.
- PrinterPort: Responsável por definir a assinatura do método que exibe os conteúdos de retorno, independente qual for a interface de saída do programa. Define que deve ter um nome de show, com apenas um argumento de tipo dinâmico, e não deve retornar nada.
- CreateConfigPort: Responsável por definir a arquitetura de um método abstrato responsável por receber um *source* arbitrário e devolver um dicionário (*Dict*).

As entidades são a representação da resolução do problema, os objetos e as instruções de como manuseá-los. Essa parte do programa não deve saber que está sendo utilizada por uma aplicação.

- Layer: representa uma camada arquitetural. Armazena os dados da prioridade, o caminho físico do *import*, e o nome virtual da camada.
- LocalConfig: representa a configuração local, isto é, as camadas e a raiz do projeto a ser analisado.
- Module: representa um módulo, contém o caminho físico e camada a qual este módulo pertence.
- BrokenRule: representa uma regra quebrada. Contém o módulo de início e o módulo final, e a linha que se encontra a inconformidade. O módulo de início é quem importou o módulo final, onde o módulo de início está em uma camada que não pode importar o módulo final.
- Report: representa um relatório, isto é, a lista de regras que foram quebradas. Contem o método add_broken responsável por adicionar da forma correta quando ocorre uma inconformidade.

5.3 Configuração e interação com o usuário

A configuração do projeto é realizada através do arquivo .calintrc.json, que deve estar na pasta raiz do projeto. Dentro da configuração, é necessário ter as seguintes entradas:

- *roots*: as raízes dos projeto. Caso tenha uma só (e.g. CALint), basta enviar como um *array* de um único item. Caso tenha várias (e.g. Python Clean Architecture Example), basta inserir todas elas.
- Cada camada possui: um nome virtual e um nome físico, assim como uma prioridade. A camada mais externa deve ser a de menor número (e.g. 0) e a camada mais interna deve ser a de maior número (e.g. 4). Um exemplo de nome virtual é "Casos de Uso", e seu nome físico correspondente *calint.use_cases*, e a prioridade com valor 3.

Ao realizar a configuração, todos os módulos dentro da pasta de cada camada estarão sendo avaliados de acordo com as regras definidas.

5.4 Técnicas de refatoração

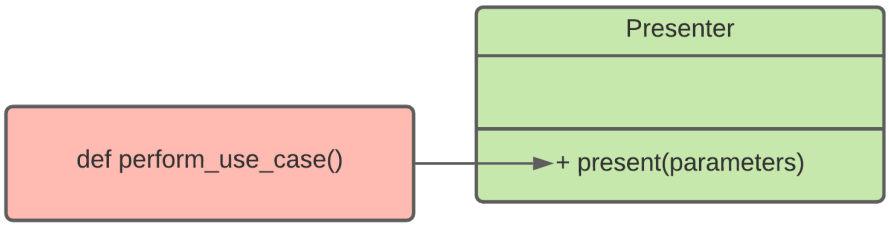
5.4.1 Adaptação de códigos existentes

A fim de refatorar as inconformidades encontradas com as regras de dependências, é necessário saber onde um módulo deve se localizar no programa e que outros módulos este pode importar. Durante os casos de estudo, foi comum encontrar que, devido a casos de uso

muito relacionados a operação de ler e escrever, as vezes é possível esquecer que não se deve chamar o adaptador diretamente dos casos de uso ou identidade. Pois ao fazer desta forma, os casos de uso passam a conhecer o adaptador, e ficam fortemente acoplados um ao outro.

A fim de solucionar essa questão, é necessário refatorar o código de forma que os adaptadores passem a depender dos casos de uso. Primeiramente, suponha que o *smell* possa ser representado como na Figura 11.

Figura 11 – Quebra da regra de dependências quando um caso de uso depende de um adaptador



A solução pode ser aplicada da seguinte forma (como foi aplicada no projeto):

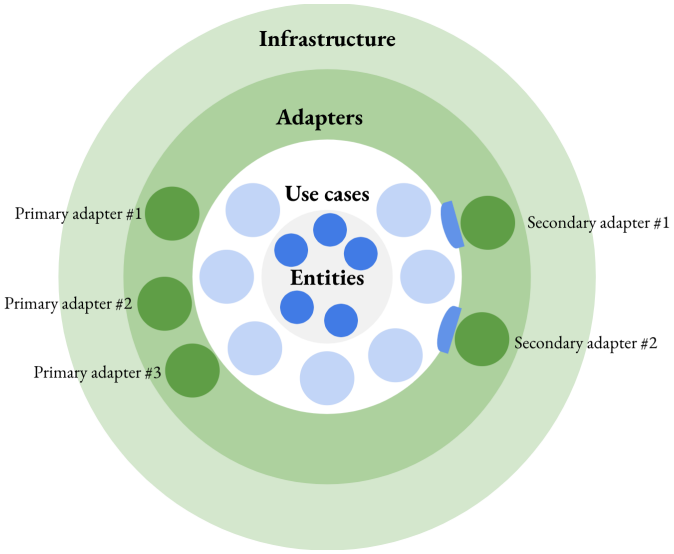
1. Verificar o que exatamente o caso de uso precisa que seja sua saída, e o que esta disponível para passar como parâmetro;
2. Verificar se essa é uma operação de entrada ou saída;
3. Criar uma assinatura de função centrada no caso de uso, com o que irá passar de parâmetro e o que deseja receber como retorno;
4. Extrair essa informação na forma de uma interface ou classe abstrata (*Extract Interface* (FOWLER, 1999)), fazer o adaptador depender ou herdar essa nova classe;
5. Substituir as chamadas a funções de adaptadores por casos de uso;
6. Injetar dependências.

5.4.2 Refatoração em programas que operam com armazenamento

Como visto em (SOARES, 2021), existem primariamente dois tipos de adaptadores, primários e secundários (Figura 12). Essa nomenclatura almeja exemplificar que alguns adaptadores tendem a se responsabilizar mais pela entrada de um programa, e outros mais de pela saída. Segundo o autor, existem três soluções técnicas (que dependem da linguagem):

1. Definir interfaces e injetar dependências;
2. Injetar funções anônimas (também conhecidas como expressões lambda);

Figura 12 – Adaptadores primários, secundários, e portas (SOARES, 2021)



3. Injetar objetos que possuem assinaturas em comum (*duck typing* e tipagem dinâmica (MILOJKOVIC; GHAFARI; NIERSTRASZ, 2017)).

Em um programa que possui repositórios ou armazenamento de dados, comumente pode ser utilizado a opção 1 (SOARES, 2021), para realizar refatoração de programas que dependem diretamente de um ORM. o ORM ou *Object Relational Mapping* se refere à um tipo de *framework* que tenta abstrair e encapsular a o código de banco de dados, mapeando os dados presentes no banco de dados para objetos na linguagem de programação utilizada (CHEN et al., 2016). Todavia, ORMs são nada mais do que *frameworks*. De forma a manter o *framework* flexível e permitir a substituição, mantendo o desacoplamento entre a aplicação e o banco de dados, é comumente utilizado a *Repository Patterns* (PRAJAPATI et al., 2019)(MICROSOFT, 2021), que pode ser vista no Python Clean Architecture Example (Capítulo 6).

A fim de refatorar um programa que acessa o ORM diretamente, é sugerido que os seguintes passos sejam realizados:

1. Agrupar os acessos pelo nome da tabela que esta sendo armazenada. Isto pois, não necessariamente todos os dados da entidade serão gravados, e gravar todos os dados da entidade faria não só que com que a entidade ficada acoplada ao mundo exterior mas também haveria perda de controle e possível vazamento de dados sensíveis (SOARES, 2021).

2. Criar uma interface utilizando a *Repository Pattern* para cada uma das tabelas. Dentro do repositório genérico, agrupar operações comuns, como pesquisar, criar, deletar, atualizar, ordenar, entre outros.

3. Fazer com que a lógica do repositório genérico seja implementada de forma concreta dentro de cada *framework*. Na parte de configuração, o *framework* então deve ser uma dependência injetada. Dessa forma, o repositório funciona é uma porta de um ou mais casos de usos.

6 Aplicações práticas e resultados

6.1 Aplicações no próprio projeto

6.1.1 Bootstrapping

Uma prática em ciência da computação é o *bootstrapping* (LOUDEN, 1997). Essa pode ser observada, por exemplo, nos compiladores. Nestes, o *bootstrapping* é a técnica pela qual um compilador recém criado é utilizado para compilar ele mesmo, com um novo código escrito na linguagem qual ele foi feito para compilar (LOUDEN, 1997).

Tendo esse conceito em mente, foi utilizado nesse projeto o próprio para guiar e verificar se a regra de dependências está sendo cumprida. Foi então criada a configuração no formato abaixo.

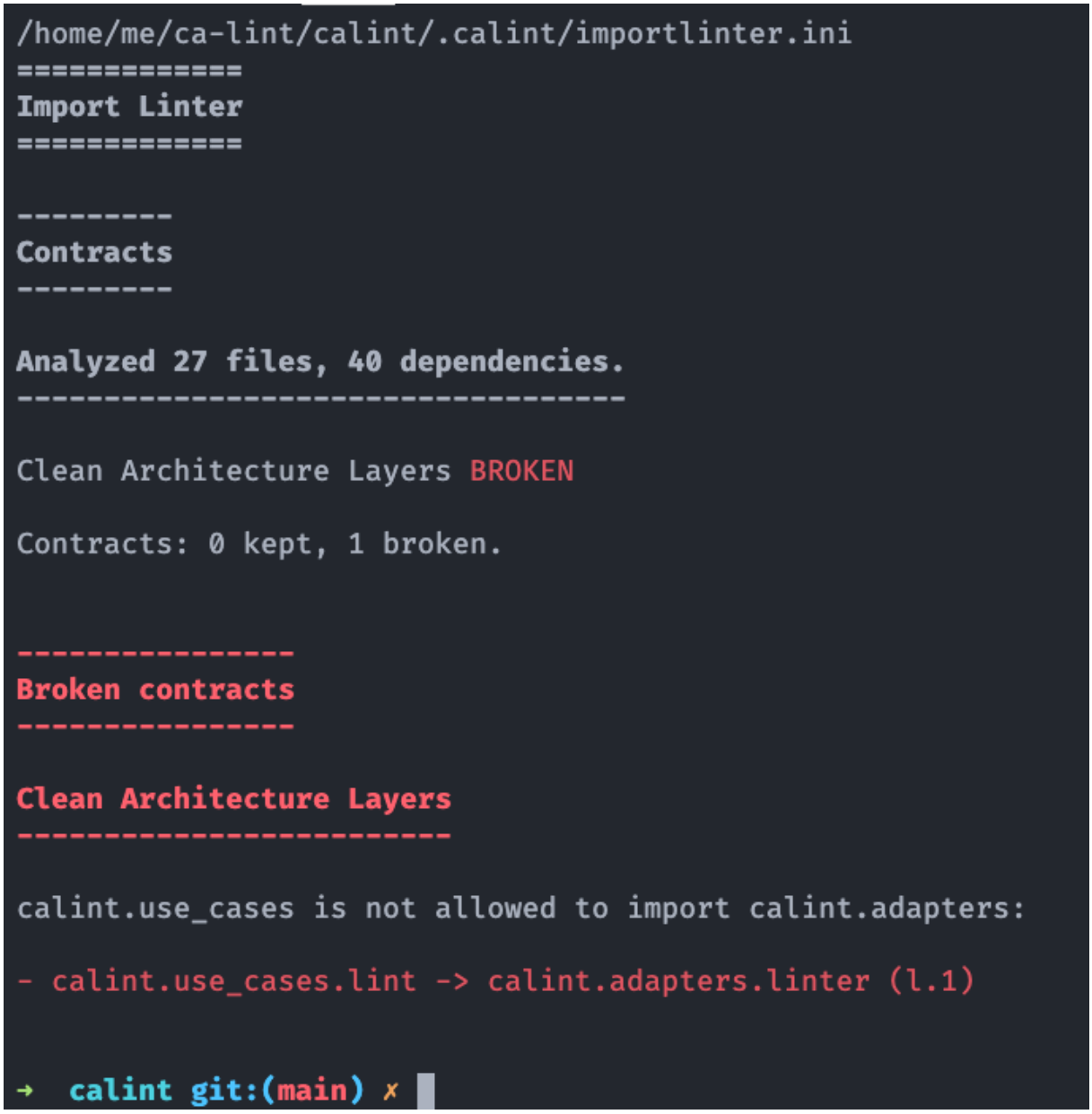
```
{
    "roots": ["calint"],
    "main": ["calint.main", 0],
    "frameworks": ["calint.frameworks", 1],
    "adapters": ["calint.adapters", 2],
    "use_cases": ["calint.use_cases", 3],
    "entities": ["calint.entities", 4]
}
```

Tendo esta estrutura em mente, as próximas subseções descrevem momentos em que houveram inconformidades.

6.1.2 Caso de uso importando um *adapter*

Em uma das versões preliminares do projeto, visto na figura 13, onde ainda não assumia a forma descrita no Capítulo 5 foi possível rodar o programa e ver que não havia sido criada uma porta pelos casos de uso, mas sim o adaptador estava guiando o desenvolvimento da lógica de negócio, uma violação visível da regra de dependências e portanto um *bad smell* da Clean Architecture.

Figura 13 – Primeira vez que foi identificado um problema de quebra da regra de dependências, antes do fim do desenvolvimento do projeto



```python
from calint.adapters.linter import LinterAdapter
from calint.use_cases.ports import LinterOptions

def lint(linter: LinterAdapter, options: LinterOptions):
    return linter.lint(options.config)
```

```python
from calint.use_cases.ports import LinterOptions, LintPort

def lint(linter: LintPort, options: LinterOptions):
    return linter.lint(options.config)
```

O código, ainda preliminar, não continha instruções muito detalhadas, pois estava passando por um experimento, onde o foco era conhecer melhor as ferramentas. Para refatorar o código visto na primeira metade, foi criado uma nova porta nos casos de uso, uma classe abstrata, que o adaptador precisou herdar para poder ser chamado pelo caso de uso após a injeção de dependências, fazendo assim com que o adaptador dependesse das instruções fornecidas pelo caso de uso, e não o contrário, como visto na segunda metade.

O mesmo problema também foi encontrado em uma versão mais próxima do final do projeto, quando o mesmo caso de uso, já mais detalhado, importava um Presenter (Figura 14) diretamente, e foi resolvido da mesma forma.

Figura 14 – Um presenter sendo importado diretamente no caso de uso

```
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) ✗
```

6.2 Aplicações em *templates* da Clean Architecture

6.2.1 Rent-o-Matic

O *Rent-o-Matic* é um projeto que pode ser descrito como uma *demo* [1] (GIORDANI, 2018) da Clean Architecture em Python. Foi criado com o fim de divulgar o livro *Clean Architectures in Python* (GIORDANI, 2016), visto que o que está no blog é o conteúdo do primeiro capítulo, segunda edição, como mencionado no fim da postagem.

6.2.1.1 Preparação

Como os nomes físicos não são os mesmos em todos os projetos, foi necessário realizar uma preparação além de apenas criar um arquivo de configuração.

O primeiro passo da preparação foi criar os arquivos __init__.py, que servem na linguagem Python para resolver os módulos (FOUNDATION, 2022b). Logo em seguida, foi necessário realizar uma re-estruturação das pastas. Haviam pastas isoladas e não separadas por módulos com o mesmo propósito. Por exemplo, havia uma pasta separada para armazenar os serializers, que poderia estar dentro de frameworks, mas estava em rest. Os elementos da pasta shared eram objetos que poderiam ser portas dos casos de uso. Entretanto, os nomes físicos foram mantidos e apenas as pastas foram realocadas.

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

6.2.1.2 Resultado

Logo em seguida, foi visto que havia uma classe abstrata vazia, que servia apenas para verificar se algo era parte do domínio, utilizando do fato que, no *Rent-o-Matic*, todos os membros do domínio deviam herdar uma classe abstrata, e afim de verificar se pertenciam ao domínio, a instância era verificada através da função nativa isinstance. Porém, esta classe se encontrava na parte de casos de uso, e o CALint foi capaz de encontrar isso imediatamente.

Para arrumar esta inconformidade, causada pela relocação das pastas, basta apenas mover essa para a pasta das entidades. Outra coisa a se observar no código é o uso do register (FOUNDATION, 2022a) para registrar essa classe como uma subclasse, mas visto que python

[1] https://github.com/lgiordani/rentomatic

Figura 15 – Entidade importando parte do caso de uso no *Rent-O-Matic*

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

1 rule broken
```

suporta múltiplas classes herdadas, basta apenas adicionar essa classe como uma das classes que todas as entidades precisam herdar. Afinal, o isintance funciona para classes herdadas também [2].

```
from abc import ABCMeta

class DomainModel(metaclass=ABCMeta):
    pass
```

```
class StorageRoom(object):
    def __init__(self, code, size, price, latitude, longitude):
        self.code = code
        self.size = size
        self.price = price
        self.latitude = latitude
        self.longitude = longitude

    # resto da classe...


DomainModel.register(StorageRoom)
```

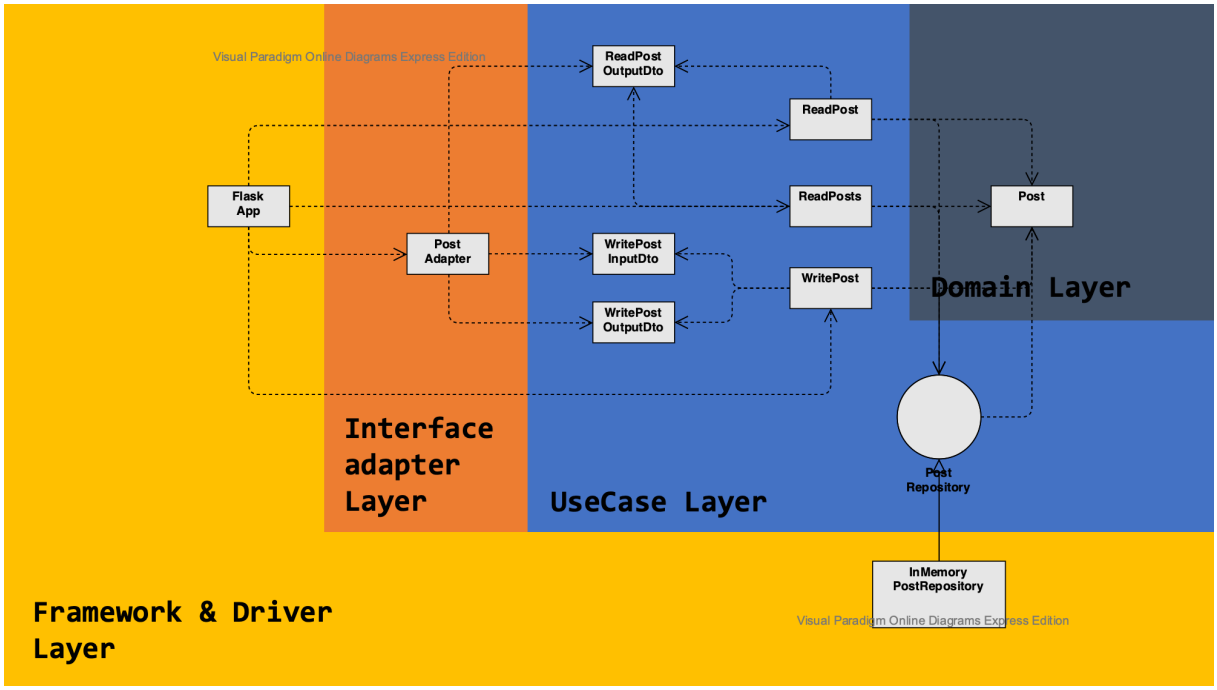
```
class StorageRoom(DomainModel):
    def __init__(self, code, size, price, latitude, longitude):
        self.code = code
        self.size = size
        self.price = price
        self.latitude = latitude
        self.longitude = longitude

    # resto da classe...
```

6.2.2 Python Clean Architecture Example

O *Python Clean Architecture Example* (HEUMSI, 2020) [3] é um projeto no *GitHub* com o objetivo de ilustrar a Clean Architecture em uma aplicação *Flask*.

[2] https://stackoverflow.com/a/1549854/
[3] https://github.com/heumsi/python-clean-architecture-example

Figura 16 – Python Clean Architecture, ilustração da arquitetura (HEUMSI, 2020)



Visto a imagem da arquitetura da aplicação, não foi necessário realizar muitas mudanças. Foi apenas criada uma pasta chamada frameworks com os módulos: main e in_memory_post_repository. Além disso, foi movido a pasta interfaces para dentro da pasta dos casos de uso, isso pois, essa pasta representa as portas. No caso desse exemplo da Clean Architecture, a arquitetura foi mantida por completo e não foi necessário gerar nenhuma mudança no código.

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

7 Conclusão

A Clean Architecture é um maneira de definir camadas de comunicação rígidas centralizando o programa em si mesmo, no lugar das *frameworks* das quais é composto. Dessa forma, é possível tornar o programa completamente configurável, já que os detalhes se tornam flexíveis o suficiente para serem re-escritos e plugados por demanda.

Por outro lado, isso requer que mais código *boilerplate* seja escrito em algumas linguagens, e que as *frameworks* tenham suas funções re-escritas para serem executadas conforme exigido pelas definições do domínio do programa. Independente de qual for a motivação para uma equipe optar pela Clean Architecture, é visível que arquitetura de software ainda possui uma escassez literária, tanto para detectar inconformidades quanto para tratá-las.

Todavia, é possível refatorar e adaptar programas existentes para a *Clean Architecture*, afinal, a maioria das refatorações envolvem princípios já bem consolidados na literatura, como inversão e injeção de dependências. Um experimento realizado na indústria se mostra necessário para averiguar e avaliar o tempo gasto para refatorar em sistemas com bases de código grandes.

Este trabalhou focou em explorar a possibilidade de refatorar os programas existentes, e foi capaz de gerar instruções passo a passo para casos mais comum. Mesmo que essas instruções utilizem de conceitos e *design patterns* já bem estabelecidas, o principal valor agregado é o de exemplificar de forma mais concreta possível como e porquê essas instruções resolvem inconformidades com a *Clean Architecture*. Visto que o resultado mais comum foi relacionado a quebra de dependências no escopo do adaptador, a checagem constante pela conformidade com a arquitetura serve como um lembrete de que esse erro está sendo cometido.

7.1 Principais contribuições

Os resultados obtidos no trabalho, servem como uma forma de divulgar novos caminhos sobre o que é a refatoração de arquiteturas. Visto que a arquitetura se encontra, de forma significativa, na regra de dependências, a análise de dependências deve se tornar uma prioridade no desenvolvimento de *software* maleável. A ferramenta sugere que é possível atender às necessidades de velocidade de desenvolvimento comumente encontradas na indústria, enquanto mantém a qualidade da arquitetura do *software*.

Para a academia, o trabalho contribui principalmente na documentação formal do estudo, e suas sugestões de refatoração realizado na linguagem Python. Com a crescente popularidade de pesquisadores utilizando Python, e a criação de novos ecossistemas utilizados na academia como o *Jupyter Notebook* (YAKIMCHIK; SHABATURA, 2019), este estudo pode servir como uma base sólida para novos projetos de pesquisa ou engenharia de *software* se ba-

searem, como um *template*, e as técnicas aqui demonstradas, como um conjunto de ferramentas básico.

O comparativo técnico do trabalho serve de base para pesquisadores e desenvolvedores que buscam diversidade nas ferramentas de qualidade de código, podendo adiantar suas necessidades ao ler este trabalho, expandindo a funcionalidade de poder de decisão da *Clean Architecture* até o propósito geral.

7.2 Trabalhos Futuros

O presente trabalho pode e deve ser ampliado, objetivando um constante aprimoramento na atividade a que se destina.

Como trabalhos futuros a esta linha de pesquisa decorrentes desta dissertação, algumas extensões podem ser feitas à abordagem para melhorar a operação do programa, adicionar funcionalidades, e integrar no desenvolvimento de *software* à nível industrial. Entre elas, destacam-se as seguintes:

Language Server Protocol: O trabalho atual serve de um *guideline* simplificado para aderir à Clean Architecture, e ainda serve como um exemplo de como implementar o rigor da Clean Architecture em Python. Todavia, o usuário ainda não possui total controle sobre o programa, e o programa não possui nenhum integração com editores de textos ou *IDEs*.

Por conta disso, como um trabalho futuro, pode ser implementado um LSP, para atender melhor os usuários durante o desenvolvimento.

Melhorias na interface de linha de comando: A interface de linha de comando atual não recebe nenhum argumento, e apenas executa o programa para a pasta atual. Como uma parte futura, seria ideal adicionar a possibilidade de indicar que pasta deve ser a pasta inicial para o projeto ser executado.

Além disso, não existe nenhum guia na interface de linha de comando que ajuda o usuário a se situar e explicar como o programa funciona. Seria interessante adicionar uma página de manual, bem como comandos de ajuda.

Checagem por configuração inválida: Atualmente o CALint não verifica se a configuração inicial é válida ou inválida, e apenas executa assumindo que a mesma seja válida. Como uma *feature* adicional, seria ideal verificar a corretura da configuração e apontar onde estão os erros.

Suporte a pacotes externos: A fim de criar um conjunto maior de regras, uma boa adição de funcionalidade futura é a de suportar incluir pacotes externos na projeto, e quais camada certos pacotes externos são proibidos de ser serem dependências.

Referências

ANDRADE, H. S. de; ALMEIDA, E.; CRNKOVIC, I. Architectural bad smells in software product lines: An exploratory study. In: *Proceedings of the WICSA 2014 Companion Volume*. [S.l.: s.n.], 2014. p. 1–6. Citado 2 vezes nas páginas 23 e 24.

BELTRãO, A.; FARZAT, F.; TRAVASSOS, G. Technical debt: A clean architecture implementation. In: *Anais Estendidos do XI Congresso Brasileiro de Software: Teoria e Prática*. Porto Alegre, RS, Brasil: SBC, 2020. p. 131–134. ISSN 2177-9384. Disponível em: <https://sol.sbc.org.br/index.php/cbsoft_estendido/article/view/14620>. Citado 4 vezes nas páginas 13, 29, 30 e 35.

BROWN, N. et al. Managing technical debt in software-reliant systems. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. [S.l.: s.n.], 2010. p. 47–52. Citado na página 35.

BROWN, W. *AntiPatterns : refactoring software, architectures, and projects in crisis*. New York: Wiley, 1998. ISBN 978-0-471-19713-3. Citado na página 20.

BUI, D. Reactive programming and clean architecture in android development. Metropolia Ammattikorkeakoulu, 2017. Citado 2 vezes nas páginas 26 e 34.

CHEN, M. et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. Citado na página 31.

CHEN, T.-H. et al. An empirical study on the practice of maintaining object-relational mapping code in java systems. In: IEEE. *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. [S.l.], 2016. p. 165–176. Citado na página 43.

COCKBURN, A. *Hexagonal architecture*. 2005. Disponível em: <https://alistair.cockburn.us-/hexagonal-architecture/>. Citado na página 18.

FERRANTE, J.; OTTENSTEIN, K. J.; WARREN, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 9, n. 3, p. 319–349, 1987. Citado na página 13.

FONTANA, F. A. et al. Arcan: A tool for architectural smells detection. In: IEEE. *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. [S.l.], 2017. p. 282–285. Citado 3 vezes nas páginas 25, 33 e 34.

FOUNDATION, E. *Language Server Protocol Explained*. 2016. Disponível em: <https:/-/www.youtube.com/watch?v=2GqpdfIAhz8>. Citado na página 14.

FOUNDATION, P. S. *Python Language Reference, Abstract Base Classes*. 2022. Disponível em: <https://docs.python.org/3/library/abc.html>. Citado na página 48.

FOUNDATION, P. S. *Python Language Reference, Modules*. 2022. Disponível em: <https://docs.python.org/3/tutorial/modules.html>. Citado na página 48.

FOWLER, M. *Refactoring : improving the design of existing code*. Reading, MA: Addison-Wesley, 1999. ISBN 978-0-201-48567-7. Citado 4 vezes nas páginas 13, 20, 25 e 42.

GAMMA, E. *Design patterns : elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995. ISBN 978-0201633610. Citado na página 25.

GARCIA, J. et al. Toward a catalogue of architectural bad smells. In: SPRINGER. *International conference on the quality of software architectures*. [S.l.], 2009. p. 146–162. Citado 4 vezes nas páginas 21, 23, 24 e 34.

GIORDANI, L. *Clean architectures in Python: a step-by-step example*. 2016. Disponível em: <https://www.thedigitalcatonline.com/blog/2016/11/14/clean-architectures-in-python-a-step-by-step-example/>. Citado na página 48.

GIORDANI, L. *Clean Architectures in Python 2nd Edition*. leanpub, 2018. Disponível em: <https://leanpub.com/clean-architectures-in-python>. Citado na página 48.

GOSEVA-POPSTOJANOVA, K.; PERHINSCHI, A. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, Elsevier, v. 68, p. 18–33, 2015. Citado na página 14.

GUO, Y. et al. Tracking technical debt — an exploratory case study. In: . [S.l.: s.n.], 2011. p. 528–531. Citado na página 13.

HAKONEN, H. et al. Improving object integrity and preventing side effects via deeply immutable references. In: CITESEER. *In Proceedings of sixth Fenno-Ugric Symposium on Software Technology, FUSST'99*. [S.l.], 1999. Citado na página 15.

HEUMSI. *Python Clean Architecture Example*. 2020. Disponível em: <https://github-.com/heumsi/python-clean-architecture-example>. Citado 3 vezes nas páginas 9, 49 e 50.

HOTTA, K.; HIGO, Y.; KUSUMOTO, S. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In: IEEE. *2012 16th European Conference on Software Maintenance and Reengineering*. [S.l.], 2012. p. 53–62. Citado na página 13.

HUGHES, J. Why functional programming matters. *The computer journal*, Oxford University Press, v. 32, n. 2, p. 98–107, 1989. Citado na página 15.

IMANKULOV, R. *Linter for Python Architecture*. 2021. Disponível em: <https://roman.pt-/posts/python-architecture-linter/>. Citado 2 vezes nas páginas 9 e 29.

JACOBSON, I. *Object-oriented software engineering : a use case driven approach*. New York Wokingham, Eng. Reading, Mass: ACM Press Addison-Wesley Pub, 1992. ISBN 0201544350. Citado na página 18.

JOHNSON, S. C. *Lint, a C program checker*. [S.l.]: Bell Telephone Laboratories Murray Hill, 1977. Citado na página 27.

KERIEVSKY, J. *Refactoring to patterns*. Boston: Addison-Wesley, 2005. ISBN 978-0321213358. Citado na página 21.

KHAN, F. et al. An empirical study of type-related defects in python projects. *IEEE Transactions on Software Engineering*, IEEE, 2021. Citado na página 30.

LIPPERT, M. *Refactoring in large software projects*. Chichester, England Hoboken, NJ: John Wiley & Sons, 2006. ISBN 9780470858936. Citado na página 23.

LOGILAB, P.; CONTRIBUTORS. *How to Write a Checker*. 2022. Disponível em: <https://pylint.pycqa.org/en/latest/how_tos/custom_checkers.html>. Citado na página 28.

LOUDEN, K. C. Compiler construction. *Cengage Learning*, 1997. Citado na página 45.

LOURIDAS, P. Static code analysis. *Ieee Software*, IEEE, v. 23, n. 4, p. 58–61, 2006. Citado na página 14.

MACCORMACK, A.; STURTEVANT, D. J. Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software*, Elsevier, v. 120, p. 170–182, 2016. Citado 3 vezes nas páginas 25, 26 e 35.

MALIK, K. I. et al. Software architecture refactoring tools and techniques: a comparative study. *Science International (Lahore)*, v. 32, n. 1, p. 27–32, 2020. Citado 2 vezes nas páginas 21 e 25.

MARTIN, I. k. R. C. *ITkonekt 2019 | Robert C. Martin (Uncle Bob), Clean Architecture and Design*. 2019. Disponível em: <https://www.youtube.com/watch?v=2dKZ-dWaCiU>. Citado 2 vezes nas páginas 19 e 37.

MARTIN, R. C. Design principles and design patterns. 2000. Disponível em: <https:/-/web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles-/Principles_and_Patterns.pdf>. Citado na página 24.

MARTIN, R. C. *Agile software development: principles, patterns, and practices*. Upper Saddle River, N.J: Prentice Hall, 2003. (Alan Apt series). ISBN 9780135974445. Citado 2 vezes nas páginas 20 e 25.

MARTIN, R. C. *Clean code : a handbook of agile software craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2009. ISBN 978-0-13-235088-4. Citado na página 21.

MARTIN, R. C. *The Clean Architecture*. 2012. Disponível em: <https://blog.cleancoder.com-/uncle-bob/2012/08/13/the-clean-architecture.html>. Citado 4 vezes nas páginas 9, 18, 19 e 20.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston, MA: Prentice Hall, 2017. (Robert C. Martin Series). ISBN 978-0-13-449416-6. Disponível em: <https://www.safaribooksonline.com/library/view/clean-architecture-a-/9780134494272/>. Citado 10 vezes nas páginas 13, 16, 18, 19, 20, 21, 24, 25, 26 e 34.

MICROSOFT. *.NET microservices - Architecture e-book*. 2021. Disponível em: <https:/-/docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns-/infrastructure-persistence-layer-design>. Citado na página 43.

MILOJKOVIC, N.; GHAFARI, M.; NIERSTRASZ, O. It's duck (typing) season! In: IEEE. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. [S.l.], 2017. p. 312–315. Citado na página 43.

MIR, A. M. et al. Type4py: Deep similarity learning-based type inference for python. *arXiv preprint arXiv:2101.04470*, 2021. Citado na página 31.

OSTROWSKI, S. *Announcing Pylance: Fast, feature-rich language support for Python in Visual Studio Code*. 2020. Disponível em: <https://devblogs.microsoft.com/python-/announcing-pylance-fast-feature-rich-language-support-for-python-in-visual-studio-code/>. Citado na página 30.

PALERMO, J. *The Onion Architecture : part 1*. 2008. Disponível em: <https://jeffreypalermo-.com/2008/07/the-onion-architecture-part-1/>. Citado na página 18.

PRAJAPATI, M. et al. Asp. net mvc-generic repository pattern and unit of work. *International Journal Of All Research Writings*, v. 1, n. 1, p. 23–30, 2019. Citado na página 43.

PRASANNA, D. *Dependency injection*. Greenwich, Conn: Manning, 2009. ISBN 9781933988559. Citado na página 25.

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*. [S.l.: s.n.], 2007. p. 7. Citado na página 39.

RHODES, B. *The Clean Architecture in Python*. 2014. Disponível em: <https://archive.org-/details/pyvideo_2840__The_Clean_Architecture_in_Python>. Citado na página 15.

RITCHIE, D. M. A stream input-output system. *UNIX Research System Papers*, p. 503–511, 1990. Citado na página 19.

RIZZI, L.; FONTANA, F. A.; ROVEDA, R. Support for architectural smell refactoring. In: *Proceedings of the 2nd International Workshop on Refactoring*. [S.l.: s.n.], 2018. p. 7–10. Citado 2 vezes nas páginas 25 e 34.

SAMARTHYAM, G.; SURYANARAYANA, G.; SHARMA, T. Refactoring for software architecture smells. In: *Proceedings of the 1st International Workshop on Software Refactoring*. [S.l.: s.n.], 2016. p. 1–4. Citado na página 21.

SOARES, L. Clean and hexagonal architectures for dummies. 2021. Disponível em: <https://medium.com/codex/clean-architecture-for-dummies-df6561d42c94>. Citado 3 vezes nas páginas 9, 42 e 43.

SURYANARAYANA, G. *Refactoring for software design smells : managing technical debt*. Amsterdam Boston: Elsevier, Morgan Kaufmann, 2015. ISBN 978-0128013977. Citado 3 vezes nas páginas 9, 21 e 22.

SZŐKE, G. et al. Faultbuster: An automatic code smell refactoring toolset. In: IEEE. *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. [S.l.], 2015. p. 253–258. Citado na página 14.

TUFANO, M. et al. When and why your code starts to smell bad. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 1, p. 403–414. Citado 2 vezes nas páginas 20 e 21.

VELASCO-ELIZONDO, P. et al. Towards detecting mvc architectural smells. In: SPRINGER. *International Conference on Software Process Improvement*. [S.l.], 2017. p. 251–260. Citado 5 vezes nas páginas 14, 25, 26, 27 e 34.

VUKASINOVIC, P. *Why cyclic dependency errors occur – a look into the Python import mechanism*. 2020. Disponível em: <https://agilno.com/why-cyclic-dependency-errors-occur-a-look-into-the-python-import-mechanism/>. Citado na página 21.

WALKER, A.; DAS, D.; CERNY, T. Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences*, Multidisciplinary Digital Publishing Institute, v. 10, n. 21, p. 7800, 2020. Citado na página 14.

WHEELER, D. J. The use of sub-routines in programmes. In: *Proceedings of the 1952 ACM national meeting (Pittsburgh)*. [S.l.: s.n.], 1952. p. 235–236. Citado na página 15.

WIN, B. D. et al. On the importance of the separation-of-concerns principle in secure software engineering. In: CITESEER. *Workshop on the Application of Engineering Principles to System Security Design*. [S.l.], 2002. p. 1–10. Citado na página 24.

YAKIMCHIK, A.; SHABATURA, S. About scientific computing within python and jupyter notebook. In: EUROPEAN ASSOCIATION OF GEOSCIENTISTS & ENGINEERS. *18th International Conference on Geoinformatics-Theoretical and Applied Aspects*. [S.l.], 2019. v. 2019, n. 1, p. 1–5. Citado na página 51.