Детектор Плагиата v. 2941 - Отчёт оригинальности: 19.01.2026 12:59:29

Проанализированный документ: Наконечний.pdf Лицензия: ВОЛОДИМИР МАТІЄВСЬКИЙ

Тип поиска: Поиск переписанного Язык: Uk

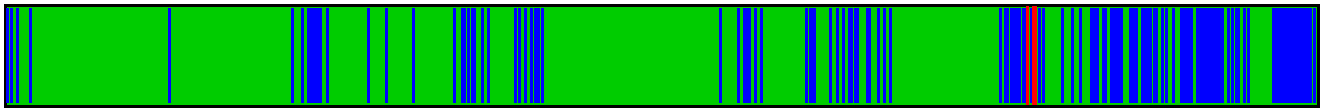
Тип проверки: Интернет

TEE и кодировка: PdfPig

Детальный анализ тела документа:

Диаграмма соотношения частей:

Граф распределения зон:



Источники плагиата: 8

6%	770	1.	https://web-crawler.plagiarism-detector.com/get-doc-pt?did=8UilP4y9-jpDkQ
5%	703	2.	https://web-crawler.plagiarism-detector.com/get-doc-pt?did=8UiiP4y7-zZBnw
1%	154	3.	https://web-crawler.plagiarism-detector.com/get-doc-pt?did=_0ilP4W79jFBlQ

Детали обработанных ресурсов: 144 - OK / 7 - Ошибок

Важные замечания:

Википедия:	Google Книги:	Сервисы платных работ:	Античит:
[не обнаружено]	[не обнаружено]	[не обнаружено]	**Обнаружено сокрытие!**

Античит-отчет UACE:

1. Статус: Анализатор **Включен** Нормализатор **Включен** сходство символов установлено на **100%**
2. Обнаруженный процент загрязнения UniCode: **21,6%** с лимитом: 4%
3. Процент нераспознанных символов после нормализации: **12,3%**
4. Все подозрительные символы будут отмечены фиолетовым цветом: Abcd...
5. Найдены невидимые символы: 0

Рекомендации по оценке:

Особое внимание следует уделить анализу этого отчета! Предполагается, что этот документ содержит значительное количество символов, чуждых языку документа. Это прямое указание на то, что автор документа использовал специальное программное обеспечение\онлайн-веб-сервис, чтобы эффективно скрыть текст в попытке избежать обнаружения потенциального плагиата. Настоятельно рекомендуется

передать это дело на более высокий уровень! В случае сомнений обращайтесь: в службу поддержки Детектора плагиата!

Алфавитная статистика и анализ символов:

Активные ссылки (URL-адреса, извлеченные из документа):

URL не найдены

Исключённые ресурсы:

URL не найдены

Включённые ресурсы:

URL не найдены

Детальный анализ документа:

Міністерство освіти і науки України Державний заклад

Цитирования: **0,04%** id: **1**

«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій Кафедра інформаційних технологій та систем Наконечний Вадим Едуардович АВТОМАТИЗОВАНА СИСТЕМА ДЕТЕКЦІЇ АРХІТЕКТУРНИХ ПОРУШЕНЬ У PYTHON-ПРОЄКТАХ НА ОСНОВІ АБСТРАКТНИХ СИНТАКСИЧНИХ ДЕРЕВ кваліфікаційна робота здобувача вищої освіти другого (магістерського) рівня освітньої програми

Цитирования: **0,01%** id: **2**

«Мультимедійні системи»

за спеціальністю 121

Цитирования: **0,02%** id: **3**

«Інженерія програмного забезпечення»

Особистий підпис _____ Вадим НАКОНЕЧНИЙ Науковий керівник _____, Володимир ДОНЧЕНКО, старший викладач кафедри інформаційних технологій та систем Завідувач кафедри _____ Микола СЕМЕНОВ, кандидат педагогічних наук, доцент кафедри інформаційних технологій та систем Лубни – 2026 АНОТАЦІЯ Тема: Автоматизована система детекції архітектурних порушень у Python- проєктах на основі абстрактних синтаксичних дерев Спеціальність: F2

Цитирования: **0,02%** id: **4**

«Інженерія програмного забезпечення».

Установа: ЛНУ імені Тараса Шевченка, 2026 р. Магістерська робота містить: 85 с., 15 рис., 1 табл., 30 джерел. Об’єктом дослідження є процес аналізу архітектури програмного забезпечення у Python-проєктах. Предметом дослідження є методи та алгоритми автоматизованої детекції архітектурних порушень на основі абстрактних синтаксичних дерев програмного коду Python. Мета дослідження – розробка та дослідження автоматизованої системи детекції архітектурних порушень у Python-проєктах на основі аналізу абстрактних синтаксичних дерев. Результати дослідження – розроблена система детекції архітектурних порушень у Python-проєктах на основі аналізу абстрактних синтаксичних дерев із формалізацією архітектурних правил та їх перевіркою на рівні структурних елементів коду. Ключові слова: ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, АРХІТЕКТУРНІ ПОРУШЕННЯ, АБСТРАКТНІ СИНТАКСИЧНІ ДЕРЕВА (AST), АВТОМАТИЗОВАНА ДЕТЕКЦІЯ, АНАЛІЗ АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ABSTRACT Topic: Automated System for Detecting Architectural Violations in Python Projects Based on Abstract Syntax Trees Speciality: F2

Цитирования: **0,01%** id: **5**

"Software engineering".

Institution: Luhansk Taras Shevchenko National University (LTSNU), 2026. Master's thesis consists of: 85 pp., 15 im., 1 tables, 30 sources Object of the study – the process of software architecture analysis in Python projects. Subject of the study – methods and algorithms for automated detection of architectural violations based on abstract syntax trees of Python source code. Purpose of the research is development and investigation of an automated system for detecting architectural violations in Python projects through the analysis of abstract syntax trees. Research results: a system was developed for detecting architectural violations in Python projects based on abstract syntax tree analysis, with formalization of architectural rules and their verification at the level of structural code elements. Keywords: SOFTWARE ENGINEERING, ARCHITECTURAL VIOLATIONS, ABSTRACT SYNTAX TREES (AST), AUTOMATED DETECTION, SOFTWARE ARCHITECTURE ANALYSIS ЗМІСТ ВСТУП 5 РОЗДІЛ 1 8 АНАЛІЗ АРХІТЕКТУРНИХ ПІДХОДІВ ТА МЕТОДІВ СТАТИЧНОГО АНАЛІЗУ PYTHON- ПРОЄКТІВ 8 1.1 8 1.2. Особливості архітектурного проєктування та аналізу коду в мові Python 12 1.3. Порівняльний аналіз підходів до статичного аналізу програмного коду 15 1.4. Огляд існуючих інструментів аналізу якості та архітектури Python-коду 17 РОЗДІЛ 2 19 МОДЕЛІ ТА АЛГОРИТМИ ВИЯВЛЕННЯ АРХІТЕКТУРНИХ ПОРУШЕНЬ НА ОСНОВІ AST 19 2.1. Представлення Python-проєкту у вигляді орієнтованого графа залежностей 19 2.2. Алгоритмічні засади аналізу

абстрактних синтаксичних дерев (AST) Python-коду 27 2.3. Формалізація архітектурних правил і механізми їх перевірки 31 2.4. Метрики оцінювання архітектурної якості та порушень 34 РОЗДІЛ 3 39 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ АВТОМАТИЗОВАНОЇ СИСТЕМИ ARCHCON 39 3.1. Архітектура та компоненти автоматизованої системи Archcon 39 3.2. Обґрунтування вибору програмних засобів і бібліотек реалізації 46 3.3. Реалізація ключових класів та логіки системи Archcon 52 3.4. Інтерфейс користувача та інтеграція системи у процес розробки 59 ВИСНОВКИ 61 СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 64 ДОДАТКИ 66 ВСТУП В умовах зростання складності програмних систем та активного використання мови Python у корпоративній розробці, наукових дослідженнях і стартап-проєктах особливої актуальності набуває проблема забезпечення та контролю архітектурної якості програмного забезпечення. Python широко застосовується для створення веб-застосунків, мікросервісів, систем аналізу даних і машинного навчання, проте його динамічна типізація, гнучка модель імпортів і слабко формалізовані архітектурні обмеження ускладнюють автоматичний контроль дотримання архітектурних рішень. У процесі еволюції програмних систем це часто призводить до накопичення архітектурних порушень, зростання технічного боргу, зниження підтримуваності та підвищення вартості супроводу програмного забезпечення. Тому розробка автоматизованих засобів виявлення архітектурних порушень у Python-проєктах є актуальним і практично значущим науково-прикладним завданням. Теоретичною базою дослідження стали фундаментальні праці з архітектури програмного забезпечення, зокрема роботи Л. Басса, П. Клементса та Р. Казмана [15], дослідження М. Фаулера у сфері рефакторингу та архітектурних антипатернів [11], а також підходи до статичного аналізу коду, запропоновані Р. Мартіном і Н. Фаулером [12]-[13]. Значний внесок у розвиток інструментів автоматичного аналізу коду зробили розробники та спільноти таких програмних продуктів і бібліотек, як Pylint, Flake8, SonarQube, Radon, а також компанії SonarSource та JetBrains, які активно досліджують питання якості та архітектурної узгодженості програмних систем. Водночас наявні інструменти орієнтовані переважно на стилістичні, синтаксичні або метрик-орієнтовані перевірки й мають обмежені можливості щодо формалізованої детекції архітектурних порушень саме на рівні структури Python-проєктів. У загальному вигляді проблема полягає у відсутності ефективних і гнучких засобів автоматизованого контролю дотримання архітектурних обмежень у Python-проєктах, які могли б аналізувати структуру коду незалежно від його виконання та виявляти порушення взаємодії між модулями, шарами та компонентами системи. Існуючі підходи або не враховують специфіку мови Python, або потребують значних зусиль для адаптації до конкретної архітектури проєкту, що знижує їх практичну ефективність у реальних умовах розробки. Метою магістерської роботи є розробка та дослідження автоматизованої системи детекції архітектурних порушень у Python-проєктах на основі аналізу абстрактних синтаксичних дерев. Для досягнення поставленої мети у роботі було визначено такі завдання: ● проаналізувати поняття архітектурних порушень і особливості їх прояву у Python-проєктах; ● дослідити існуючі підходи та інструменти статичного аналізу коду; обґрунтувати доцільність використання абстрактних синтаксичних дерев для архітектурного аналізу; спроєктувати архітектуру автоматизованої системи; ● реалізувати основні алгоритми аналізу AST та виявлення архітектурних порушень; ● провести тестування системи та оцінити її ефективність. Об’єктом дослідження є процес аналізу архітектури програмного забезпечення у Python-проєктах. Предметом дослідження є методи та алгоритми автоматизованої детекції архітектурних порушень на основі абстрактних синтаксичних дерев програмного коду Python. У ході виконання роботи були використані методи теоретичного аналізу й узагальнення наукових джерел, методи статичного аналізу програмного коду, елементи теорії графів для представлення залежностей між модулями, методи об’єктно-орієнтованого та компонентного проєктування, а також експериментальні методи оцінювання ефективності програмних засобів. Наукова новизна роботи полягає у розробці підходу до детекції архітектурних порушень у Python-проєктах на основі аналізу абстрактних синтаксичних дерев із формалізацією архітектурних правил та їх перевіркою на рівні структурних елементів коду. Практична цінність отриманих результатів полягає у можливості використання розробленої системи в процесі розробки та супроводу Python-проєктів для автоматичного контролю архітектурної якості, зменшення технічного боргу та підвищення підтримуваності програмного забезпечення. У першому розділі роботи здійснено аналіз теоретичних основ архітектури програмного забезпечення, розглянуто типові архітектурні порушення та особливості їх виникнення у Python-проєктах, а також проведено огляд існуючих інструментів статичного аналізу. У другому розділі

представлено проєктування автоматизованої системи детекції архітектурних порушень, описано модель аналізу на основі абстрактних синтаксичних дерев і загальну архітектуру системи. У третьому розділі наведено реалізацію розробленої системи, результати її тестування на реальних Python-проєктах та оцінювання ефективності запропонованого підходу. РОЗДІЛ 1 АНАЛІЗ АРХІТЕКТУРНИХ ПІДХОДІВ ТА МЕТОДІВ СТАТИЧНОГО АНАЛІЗУ PYTHON-ПРОЄКТІВ 1.1 Архітектурні патерни програмного забезпечення та типові порушення їх реалізації Архітектурні патерни програмного забезпечення визначають загальні принципи організації системи, правила взаємодії між її компонентами та допустимі залежності між ними. Вони відіграють ключову роль у забезпеченні масштабованості, підтримуваності та еволюційності програмних систем. У процесі розвитку проєктів, особливо написаних мовами з динамічною типізацією, зокрема Python, дотримання обраного архітектурного патерну часто порушується, що призводить до деградації архітектури та накопичення технічного боргу [2]. Рис. 1.1 Патерн Layered Architecture з можливим порушенням. Одним із найпоширеніших архітектурних патернів є Layered Architecture (шарова архітектура), яка передбачає поділ системи на логічні шари, наприклад, представлення, бізнес-логіки та доступу до даних (рис. 1.1). Основною вимогою цього патерну є односпрямованість залежностей між шарами [11]. Типовими порушеннями є прямі виклики між віддаленими шарами, зворотні залежності або доступ шару представлення безпосередньо до шару збереження даних. У Python- проєктах такі порушення виникають особливо часто через відсутність жорстких механізмів контролю залежностей та вільне використання імпортів між модулями. Рис. 1.2 Патерн Clean Architecture Clean Architecture (рис. 1.2), запропонована Robert C. Martin [12], є розвитком ідей шарової архітектури та ґрунтується на принципі інверсії залежностей, за яким внутрішні шари не повинні залежати від зовнішніх. Для мови Python ця архітектура є концептуально придатною, однак на практиці її реалізація часто ускладнюється через динамічну типізацію та відсутність формалізованих інтерфейсів. Типові порушення Clean Architecture у Python- проєктах включають пряме використання фреймворків у доменній логіці, залежності бізнес-правил від інфраструктурних компонентів і змішування відповідальностей у модулях. Рис. 1.3. MVC Патерн MVC (Model–View–Controller) широко застосовується у веб- розробці та підтримується багатьма Python-фреймворками (рис. 1.3) . Він передбачає чіткий розподіл відповідальностей між моделлю, поданням і контролером. Найпоширенішими порушеннями є перенесення бізнес-логіки до контролерів або подань, прямий доступ подань до джерел даних та надмірне зростання складності контролерів. У Python-проєктах такі порушення часто маскуються простотою синтаксису та відсутністю строгих обмежень на структуру класів і функцій [15]. Hexagonal Architecture (Ports and Adapters) орієнтована на ізоляцію бізнес- логіки від зовнішніх систем через порти та адаптери (рис. 1.4) [17]. Для Python цей підхід є доволі перспективним, особливо у сервісно-орієнтованих і тестованих системах. Водночас типовими порушеннями є пряме використання зовнішніх бібліотек у доменних модулях, відсутність чітко виділених портів та змішування ролей адаптерів і ядра системи. Такі порушення значно знижують тестованість і ускладнюють заміну інфраструктурних компонентів. Рис. 1.4 Hexagonal Architecture Патерн Microkernel (Plug-in Architecture) використовується для систем, які потребують розширюваності та підтримки плагінів. У Python він реалізується відносно просто завдяки механізмам динамічного імпорту та рефлексії. Водночас типовими порушеннями є тісне зв’язування ядра з плагінами, відсутність чітких контрактів між ними та неконтрольовані залежності, що суперечить ідеї мінімального ядра (рис. 1.5) [4]. Загалом, для Python-проєктів найбільш придатними є Layered Architecture, MVC, Clean Architecture та Hexagonal Architecture, оскільки вони добре поєднуються з модульною структурою мови та її екосистемою. Водночас гнучкість Python, яка є його перевагою, значно підвищує ризик архітектурних порушень. Це зумовлює необхідність автоматизованих засобів аналізу, здатних виявляти відхилення від обраного архітектурного патерну на рівні структури коду, що й обґрунтовує доцільність використання абстрактних синтаксичних дерев для детекції таких порушень. 1.2. Особливості архітектурного проєктування та аналізу коду в мові Python Мова програмування Python орієнтована на простоту, гнучкість і швидкість розробки, що безпосередньо впливає на підходи до архітектурного проєктування програмних систем. На відміну від мов зі статичною типізацією та жорстко визначеними механізмами інкапсуляції, Python не нав’язує формалізованої архітектурної структури, залишаючи питання організації коду та контролю залежностей на рівні домовленостей між розробниками. Така свобода сприяє швидкому прототипуванню, проте у великих і довготривалих проєктах суттєво

підвищує ризик архітектурних порушень [20]. Однією з ключових особливостей Python є динамічна типізація, яка ускладнює статичний аналіз архітектури програмного забезпечення. Відсутність обов’язкових типових контрактів між компонентами системи призводить до того, що архітектурні залежності часто не мають явного опису в коді. Це ускладнює виявлення порушень принципів інверсії залежностей, ізоляції доменної логіки та розмежування відповідальностей між архітектурними шарами, особливо за відсутності анотацій типів [20]. Характерною рисою Python є так званий

Цитирования: **0,01%** id: **6**

“pythonic way”

організації проєкту, відповідно до якого структура каталогів і пакетів безпосередньо відображає архітектуру системи. Використання файлів `_ _init__.py` дозволяє об’єднувати модулі в пакети, які часто відповідають архітектурним шарам або підсистемам, наприклад `app/domain`, `app/application`, `app/adapters` або `infrastructure`. Такий підхід робить архітектуру візуально зрозумілою та легкою для навігації. Водночас Python не має вбудованих механізмів обмеження доступу між пакетами, аналогічних модифікаторам `private` або `internal` у мовах зі статичною типізацією. У результаті будь-який модуль може імпортувати будь-який інший, що створює сприятливі умови для порушення меж між архітектурними шарами та появи несанкціонованих залежностей. Особливу складність для архітектурного аналізу створює модель імпортів у Python. Мова дозволяє виконувати абсолютні та відносні імпорти, умовні імпорти, імпорти всередині функцій і методів, а також динамічні імпорти. Це ускладнює побудову повного та коректного графа залежностей між модулями й підвищує ймовірність виникнення циклічних залежностей, які важко виявити без спеціалізованих інструментів аналізу. Окрему групу архітектурних викликів становлять механізми метапрограмування та динамічної модифікації коду, притаманні Python. Такі можливості, як monkey patching, використання декораторів, метакласів і динамічних атрибутів, дозволяють реалізовувати складні архітектурні патерни, зокрема Dependency Injection або Aspect-Oriented Programming, у дуже лаконічній формі. Проте ці механізми створюють неявні залежності між компонентами, які не завжди можна виявити шляхом простого аналізу тексту програми. Значна частина взаємодій між компонентами формується опосередковано, що ускладнює контроль архітектурної цілісності системи. Аналіз Python-коду ускладнюється також низкою фундаментальних обмежень статичного аналізу. По-перше, у Python відсутній окремий етап компіляції, під час якого могли б виявлятися архітектурні помилки, тому значна частина проблем проєктування проявляється лише на етапі тестування або безпосередньо в експлуатації. По-друге, без використання анотацій типів складно визначити, які саме об’єкти передаються між архітектурними компонентами, що обмежує точність аналізу взаємодії між шарами системи. Водночас Python надає потужні засоби для аналізу структури програмного коду за допомогою абстрактних синтаксичних дерев. Стандартний модуль `ast` дозволяє отримати формальне представлення програми у вигляді дерева, яке відображає всі основні синтаксичні конструкції, включаючи імпорти, визначення класів і функцій, виклики та вкладені структури. Аналіз AST дає змогу дослідити архітектуру системи так, як її сприймає інтерпретатор, незалежно від фактичного виконання коду, що є особливо важливим у контексті динамічної природи Python. Таким чином, архітектурне проєктування та аналіз коду в Python характеризуються поєднанням високої гнучкості, слабкої формалізації архітектурних обмежень і значної кількості неявних залежностей. Саме ці особливості обґрунтовують доцільність застосування аналізу абстрактних синтаксичних дерев як основи для автоматизованої системи детекції архітектурних порушень, що дозволяє підвищити контроль за дотриманням архітектурних рішень у Python-проєктах. 1.3. Порівняльний аналіз підходів до статичного аналізу програмного коду Статичний аналіз коду є одним із ключових підходів до забезпечення якості програмного забезпечення, оскільки дозволяє виявляти потенційні помилки, стилістичні відхилення та структурні проблеми без виконання програми. В екосистемі Python існує значна кількість інструментів статичного аналізу, однак більшість із них орієнтовані на локальні властивості коду або метрики якості, а не на формалізований контроль архітектурних обмежень на рівні всієї системи [20]. Одним із найпоширеніших інструментів є Pylint, [25] який здійснює глибокий аналіз коду на основі абстрактних синтаксичних дерев. Pylint перевіряє відповідність коду стандартам стилю, виявляє потенційні помилки, проблеми з іменуванням, невикористані змінні та деякі типові антипатерни. Хоча Pylint використовує AST, його основний фокус зосереджений на якості окремих модулів і класів, а

не на аналізі архітектурних залежностей між пакетами або шарами. Можливості конфігурації правил обмежені і не дозволяють формалізувати складні архітектурні інваріанти, наприклад заборону залежностей між певними підсистемами. Іншим популярним інструментом є Flake8, який поєднує декілька аналізаторів і зосереджується переважно на перевірці стилю та простих помилок. Flake8 працює швидко та легко інтегрується в процес розробки, проте має ще більш обмежені можливості з точки зору архітектурного аналізу. Він практично не враховує структурні аспекти проєкту, а його розширення за допомогою плагінів здебільшого спрямоване на локальні перевірки коду, а не на аналіз міжмодульних залежностей. Для більш комплексного аналізу програмного забезпечення широко використовується платформа SonarQube [27], яка підтримує Python та інші мови програмування. SonarQube надає розширений набір метрик якості, виявляє code smells, дублювання коду та оцінює технічний борг. Платформа дозволяє аналізувати проєкт у цілому, однак її архітектурні можливості обмежені загальними правилами та шаблонами. Формалізація специфічних архітектурних правил для конкретного Python-проєкту потребує значних зусиль або взагалі є неможливою без розробки власних плагінів. Окрему категорію становлять інструменти аналізу метрик складності, зокрема Radon, які оцінюють цикломатичну складність, рівень вкладеності та інші кількісні показники. Такі інструменти корисні для оцінювання якості коду, але не дають уявлення про відповідність реалізації обраній архітектурі. Вони не аналізують логічні межі між компонентами та не здатні виявляти порушення архітектурних шарів або інверсії залежностей. У контексті архітектурного аналізу варто відзначити підходи, реалізовані в інструментах типу ArchUnit для інших мов програмування, які дозволяють описувати архітектурні правила у вигляді декларативних обмежень. Проте для Python відсутні зрілі ArchUnit-подібні бібліотеки, які б надавали гнучкі механізми формалізації та автоматичної перевірки архітектурних інваріантів на основі структури коду. Таким чином, аналіз існуючих інструментів статичного аналізу коду Python показує, що вони ефективні для виявлення стилістичних помилок, локальних дефектів і загальних проблем якості, але не забезпечують повноцінної підтримки архітектурного аналізу. Це створює розрив між задекларованою архітектурою системи та її фактичною реалізацією. Виявлений недолік обґрунтовує необхідність розробки спеціалізованої автоматизованої системи, здатної аналізувати Python-проєкти на основі абстрактних синтаксичних дерев і формалізованих архітектурних правил, що й становить основну ідею системи Archcon. 1.4. Огляд існуючих інструментів аналізу якості та архітектури Python-коду Забезпечення якості програмного забезпечення в Python-проєктах традиційно ґрунтується на використанні інструментів статичного аналізу, які дозволяють виявляти помилки, порушення стилю та окремі дефекти проєктування. Проте більшість наявних засобів орієнтовані насамперед на якість коду на мікрорівні й лише частково або опосередковано торкаються питань архітектури програмної системи в цілому. Одним із найпоширеніших інструментів аналізу Python-коду є Pylint [25]. Він виконує аналіз коду на основі абстрактних синтаксичних дерев і надає детальні звіти щодо стилістичних порушень, потенційних помилок, складності коду та відповідності рекомендаціям PEP 8. Хоча Pylint дозволяє створювати власні перевірки, його архітектурні можливості є обмеженими: інструмент не призначений для опису міжмодульних або міжшарових обмежень і не підтримує декларативне задання архітектурних правил. У результаті Pylint фокусується переважно на локальних властивостях коду, а не на цілісності архітектури системи. Інструмент Flake8 [26] є ще більш орієнтованим на перевірку стилю та базових помилок. Він поєднує декілька аналізаторів і широко використовується у процесах безперервної інтеграції завдяки простоті налаштування та високій швидкодії. Водночас Flake8 практично не аналізує структуру проєкту, не будує графів залежностей і не надає механізмів для контролю архітектурних рішень. Його розширюваність за допомогою плагінів здебільшого застосовується для додаткових стилістичних або синтаксичних перевірок, а не для архітектурного аналізу. Для комплексної оцінки якості програмного забезпечення часто використовується платформа SonarQube, яка підтримує аналіз Python-проєктів у межах єдиного середовища для різних мов програмування. SonarQube надає широкий набір метрик, виявляє code smells, дублювання коду та оцінює рівень технічного боргу. Однак архітектурний аналіз у цій платформі реалізований на узагальненому рівні та орієнтований переважно на типові шаблони проблем. Налаштування кастомних архітектурних правил для конкретного Python-проєкту є складним або потребує розробки власних плагінів, що суттєво ускладнює практичне використання SonarQube як інструменту контролю архітектурних інваріантів. Окрему увагу заслуговують так звані ArchUnit-like підходи, які добре зарекомендували себе в інших мовах програмування завдяки

можливості декларативного опису архітектурних правил. Такі інструменти дозволяють формалізувати обмеження на залежності між пакетами, шарами та компонентами системи. Проте для Python наразі відсутні зрілі та широко використовувані бібліотеки цього класу. Існуючі експериментальні або нішеві рішення або мають обмежений функціонал, або потребують значних зусиль для інтеграції та підтримки, що знижує їхню привабливість для промислового використання. Таким чином, огляд існуючих інструментів аналізу якості та архітектури Python-коду показує, що вони ефективно вирішують завдання контролю стилю, пошуку локальних дефектів і оцінювання метрик якості, проте не забезпечують повноцінної підтримки архітектурного аналізу. Основними їх недоліками є фокус на синтаксичних і стилістичних аспектах замість архітектурних, обмежені можливості опису кастомних архітектурних правил і складність контролю міжмодульних залежностей. Це обґрунтовує необхідність розробки спеціалізованої системи, здатної формалізувати архітектурні вимоги та автоматично перевіряти їх дотримання у Python-проєктах, що й визначає актуальність запропонованого в роботі підходу. РОЗДІЛ 2 МОДЕЛІ ТА АЛГОРИТМИ ВИЯВЛЕННЯ АРХІТЕКТУРНИХ ПОРУШЕНЬ НА ОСНОВІ AST 2.1. Представлення Python-проєкту у вигляді орієнтованого графа залежностей Для опису

Цитирования: **0,01%** id: **7**

«правильної»

архітектури в межах розробки системи

Цитирования: **0,01%** id: **8**

«Archcon»

необхідно впровадити механізм формалізації архітектурних обмежень, який трансформує абстрактні концепції проєктування у машиночитний формат. Найбільш доцільним підходом є використання декларативних мов розмітки, таких як YAML або JSON, що дозволяють визначити топологію системи через набір правил та обмежень. Процес формалізації починається з ідентифікації архітектурних компонентів та їхнього відображення на фізичну структуру каталогів і файлів Python-проєкту. Кожен рівень (layer) або модуль описується як логічна одиниця, для якої встановлюється політика допустимих зв'язків. Центральним елементом такої конфігурації є матриця доступу або список заборон (denylist), що базується на принципі інверсії залежностей та ієрархічності. Наприклад, для реалізації Clean Architecture у схемі YAML визначається кореневий вузол

Цитирования: **0,01%** id: **9**

«шари»,

де для компонента `Domain` встановлюється властивість `forbidden_dependencies`, що включає `Infrastructure` та `Presentation`. Це дозволяє системі стабілізувати ядро застосунку, запобігаючи витоку технічних деталей у бізнес-логіку. Окрім прямих заборон, конфігураційна схема може підтримувати регулярні вирази для мапування пакетів, що забезпечує гнучкість при масштабуванні проєкту. Таблиця 2.1 – приклад матриці доступу Від (Source) \ До Use Domain Adapters Infrastructure (Target) Cases Domain + - - - Use Cases + + - - Adapters + + + - Infrastructure + + + + У табл. 2.1

Цитирования: **0,01%** id: **10**

«+»

значить дозволено: шар може імпортувати компоненти вказаного рівня.;

Цитирования: **0,01%** id: **11**

«-»

значить заборонено: спроба імпорту призведе до фіксації архітектурного порушення системою Archcon. Список заборон представимо у форматі YAML у листингу 2.1 Листинг 2.1 – список заборон # Archcon Configuration: Architecture Guard Rules version: 1.0 project_name:

Цитирования: **0,01%** id: **12**

"MyPythonProject"

layers: domain:

Цитирования: **0,03%** id: **13**

"app.core.domain.*"

use_cases:

Цитирования: **0,03%** id: **14**

"app.core.use_cases.*"

adapters:

Цитирования: **0,02%** id: **15**

"app.adapters.*"

infrastructure:

Цитирования: **0,02%** id: **16**

"app.infrastructure.*"

rules: # Правила для доменного шару (найсуворіші) - layer:

Цитирования: **0,01%** id: **17**

"domain"

forbidden_dependencies: -

Цитирования: **0,01%** id: **18**

"use_cases"

-

Цитирования: **0,01%** id: **19**

"adapters"

-

Цитирования: **0,01%** id: **20**

"infrastructure"

message:

Цитирования: **0,07%** id: **21**

"Критична помилка: Доменний шар не може залежати від зовнішніх рівнів."

Правила для випадків використання - layer:

Цитирования: **0,01%** id: **22**

"use_cases"

forbidden_dependencies: -

Цитирования: **0,01%** id: **23**

"adapters"

-

Цитирования: **0,01%** id: **24**

"infrastructure"

message:

Цитирования: **0,07%** id: **25**

"Порушення: Use Cases повинні бути ізольовані від деталей реалізації (Adapters/Infra)."

Заборона циклічних імпортів (загальне правило) - layer:

Цитирования: **0,01%** id: **26**

"*"

check_circular_dependencies: true Використання такої формалізованої моделі дозволяє відокремити логіку перевірки від самих правил, що робить систему

Цитирования: **0,01%** id: **27**

«Archcon»

універсальною. У процесі роботи системи блок валідації зіставляє отриманий у результаті аналізу AST граф фактичних імпортів із еталонною моделлю, описаною в конфігурації. Будь-яка невідповідність — наприклад, спроба доменного об’єкта імпортувати драйвер бази даних — ідентифікується як архітектурний відхил (architectural drift). Такий підхід

забезпечує відтворюваність аналізу та дозволяє інтегрувати контроль архітектурної цілісності безпосередньо в конвеєри безперервної інтеграції (CI/CD), де конфігураційний файл виступає в ролі

Цитирования: **0,01%** id: **28**

«контракту»

між архітектором та командою розробки. Для автоматизованого аналізу архітектури програмного забезпечення важливо мати формальну модель, яка дозволяє однозначно описати структуру системи та взаємозв’язки між її елементами. Однією з найбільш зручних і поширених форм такого подання є орієнтований граф залежностей, у якому вузли відповідають структурним елементам проєкту, а дуги — відношенням залежності між ними. Застосування графової моделі дозволяє перевести задачу аналізу архітектури Python-проєкту у площину формальних алгоритмів і правил, що є необхідною передумовою для автоматичної детекції архітектурних порушень. У контексті Python-проєкту вершинами орієнтованого графа можуть виступати різні рівні абстракції: пакети, модулі, класи або окремі функції. Найбільш доцільним для архітектурного аналізу є подання на рівні пакетів і модулів, оскільки саме вони зазвичай відповідають архітектурним шарам або підсистемам. Орієнтоване ребро між двома вершинами означає наявність залежності, наприклад імпорту одного модуля в іншому або використання класу чи функції з іншого пакета. Напрямок ребра відображає напрям залежності, тобто той факт, що один компонент використовує інший. Побудова такого графа для Python-проєкту ґрунтується на аналізі структури вихідного коду. Основним джерелом інформації про залежності є оператори `import` та `from ... import ...`, які явно фіксують зв’язки між модулями. Додатково можуть враховуватися виклики класів і функцій, успадкування, а також використання атрибутів інших модулів. На цьому етапі особливо важливо відрізняти внутрішні залежності проєкту від залежностей на зовнішні бібліотеки, оскільки архітектурні правила зазвичай формулюються саме для внутрішньої структури системи. Орієнтований граф залежностей дозволяє наочно відобразити ключові архітектурні властивості системи. Зокрема, на його основі можна виявляти циклічні залежності між модулями, що є типовим архітектурним дефектом, аналізувати дотримання шарової структури та перевіряти односпрямованість залежностей. Наприклад, у шаровій архітектурі допускаються лише залежності від вищих шарів до нижчих, тоді як зворотні ребра у графі свідчитимуть про архітектурні порушення. Рис. 3.1.Приклад орієнтованого граф залежностей для Python-проєкт У наведеному прикладі Python-проєкт подано як орієнтований граф залежностей, де вершини відповідають ключовим пакетам (архітектурним зонам), а дуги показують напрямок залежності (імпорт/використання). Пакет api (вхідний шар) залежить від app/application, оскільки маршрути/контролери викликають use cases. Пакет app/application залежить від app/domain, оскільки сценарії використовують доменні сутності, правила та порти. Пакет app/adapters з’єднує зовнішній світ із ядром: він може залежати від application (викликає use cases) і від domain (працює з сутностями/портами), а також взаємодіє з infrastructure (БД, черги, зовнішні API), реалізуючи конкретні технічні адаптери. Пакет utils використовується як спільний набір допоміжних функцій, від якого можуть залежати різні частини системи. Пакет tests імпортує модулі системи для тестування, тому на графі показані його залежності на api та application. Окремо додано приклад архітектурного порушення — червону пунктирну дугу app/domain → infrastructure. У Clean/Hexagonal-підходах доменний шар має бути ізольований від інфраструктурних деталей (БД, HTTP-клієнтів тощо), тому така залежність є індикатором деградації архітектури. Саме подібні заборонені ребра, цикли та

Цитирования: **0,01%** id: **29**

«неправильні напрямки»

залежностей і є типовими об’єктами детекції системи Archcon. Для складніших архітектурних стилів, таких як Clean Architecture або Hexagonal Architecture, граф залежностей дозволяє перевіряти дотримання принципу інверсії залежностей. У цьому випадку аналізується не лише наявність зв’язків, а й їх напрям: внутрішні компоненти не повинні залежати від зовнішніх. Таким чином, орієнтований граф стає основою для формалізації архітектурних інваріантів у вигляді правил над множинами вершин і ребер. Рис. 2.2. Граф залежностей: Layered Architecture На рис. 2.2 залежності йдуть строго зверху вниз. Якщо стрілка йде вгору – це порушення. Рис. 2.3. Граф залежностей: Clean Architecture На рис. 2.3 усі імпорти сходяться до центру (Entities). Зовнішні фреймворки

залежать від Use Cases, а не навпаки. Рис. 2.4. Граф залежностей: MVC (Model-View-Controller) У правильному MVC (рис. 2.4) Controller імпортує Model і View. View може імпортувати Model (для читання), але Model не знає ні про кого. Використання орієнтованого графа залежностей також спрощує інтеграцію системи аналізу з алгоритмічними методами перевірки. Архітектурні правила можуть бути представлені як обмеження на шляхи в графі, заборона певних типів ребер або вимоги до відсутності циклів. Це дозволяє реалізувати універсальний механізм перевірки архітектури, незалежний від конкретної предметної області Python-проєкту. Рис. 2.5. Граф залежностей: Hexagonal Architecture Рис. 2.5 показує інверсію залежностей. RepositoryImpl (Адаптер) імпортує Interfaces (Порт) у домені. Service (Ядро) теж імпортує Interfaces. Ніхто не імпортує RepositoryImpl напряму (його

Цитирования: **0,01%** id: **30**

"впорскують"

через DI). Таким чином, представлення Python-проєкту у вигляді орієнтованого графа залежностей є ключовим етапом побудови автоматизованої системи детекції архітектурних порушень. Воно забезпечує формальну основу для аналізу структури коду, дозволяє застосовувати алгоритмічні методи перевірки та створює передумови для гнучкого й розширюваного опису архітектурних правил у системі Archcon. 2.2. Алгоритмічні засади аналізу абстрактних синтаксичних дерев (AST) Python- коду Алгоритмічне забезпечення системи детекції архітектурних порушень у Python-проєктах ґрунтується на аналізі абстрактних синтаксичних дерев (Abstract Syntax Tree, AST), які надають формальне структурне представлення програмного коду незалежно від його виконання. На відміну від текстового аналізу, підхід на основі AST дозволяє працювати з кодом на рівні синтаксичних конструкцій, що робить можливим виявлення залежностей між модулями та компонентами системи, а також перевірку архітектурних інваріантів за формалізованими правилами. Побудова AST у Python виконується стандартними засобами мови. Кожен вихідний файл `*.py` після синтаксичного розбору перетворюється на дерево вузлів, де кожен вузол відповідає певній конструкції: імпорту, оголошенню класу чи функції, виклику, присвоєнню, умовному оператору тощо. Для задач архітектурного аналізу найбільш значущими є вузли, що описують модульну структуру проєкту та зв’язки між його частинами, насамперед `Import`, `ImportFrom`, `ClassDef`, `FunctionDef`, `Call`, `Attribute`, `Assign` і пов’язані з ними структури. Алгоритмічна модель аналізу AST у межах системи Archcon включає послідовність етапів (рис. 2.6). На першому етапі виконується підготовка вхідних даних: сканування каталогу проєкту, відбір Python-файлів та нормалізація шляхів із визначенням повних модульних імен. Далі для кожного файлу виконується синтаксичний розбір і отримується AST-представлення. На цьому етапі важливо забезпечити стійкість алгоритму до помилок коду, оскільки на практиці можуть аналізуватися неповні або тимчасово некоректні гілки репозиторію. Другий етап полягає в обході дерева та витягуванні структурних фактів. Найчастіше використовується обхід у глибину, який може бути реалізований за допомогою патерну

Цитирования: **0,01%** id: **31**

«Visitor».

Рис. 2.6. Діаграма діяльності для аналізу AST Для кожного вузла AST застосовуються правила вилучення даних: з вузлів імпорту визначаються модулі-джерела та модулі-призначення; з оголошень класів і функцій витягуються імена, простір видимості, базові класи; з вузлів викликів і доступу до атрибутів можна отримати ознаки потенційної взаємодії між модулями. Результатом цього етапу є набір фактів, що описують структуру проєкту: множина сутностей (модулі, пакети, класи, функції) та множина відношень (імпортує, наслідує, викликає, використовує). Третій етап — побудова орієнтованого графа залежностей. На основі зібраних фактів формується граф, у якому вершинами є елементи вибраного рівня абстракції (наприклад модулі або пакети), а орієнтовані ребра відповідають залежностям. У найбільш типовому сценарії для архітектурного аналізу використовується граф модульних залежностей на основі імпортів, оскільки імпорт є найстабільнішою і найменш неоднозначною ознакою взаємодії компонентів у Python. Побудований граф слугує універсальною основою для перевірки архітектурних правил і пошуку порушень, зокрема циклів, заборонених напрямків залежностей та несанкціонованих зв’язків між підсистемами. Четвертий етап пов’язаний із формалізацією та застосуванням правил. Архітектурні обмеження зручно задавати у вигляді правил над

графом: заборона ребер між певними множинами вершин, перевірка відсутності шляхів певного типу, контроль інверсії залежностей або перевірка циклічності. Для цього використовуються алгоритми теорії графів. Наприклад, для пошуку циклів застосовуються алгоритми пошуку сильно зв’язаних компонент або DFS з маркуванням; для перевірки досяжності — пошук у ширину або глибину; для контролю шарів — перевірка топологічної узгодженості залежностей відносно заданої ієрархії. Таким чином, виявлення архітектурних порушень перетворюється на задачу перевірки формальних властивостей графа. Важливим аспектом є робота з неоднозначностями, що властиві Python. Зокрема, частина залежностей може бути прихована через динамічні імпорти, використання `__import__`, `importlib`, а також метапрограмування. У таких випадках система або обмежується підмножиною залежностей, які можна надійно визначити статично, або використовує евристичні підходи, наприклад виявлення потенційних імпортів через аналіз аргументів функцій імпорту. Додатково може враховуватися структура пакетів і наявність файлів `__init__.py`, що дозволяє точніше співвідносити файлову структуру з модульними іменами. Отже, алгоритмічні засади аналізу AST Python-коду в системі Archcon базуються на послідовному перетворенні вихідного коду у формальне деревоподібне представлення, вилученні структурних фактів, побудові графа залежностей та перевірці архітектурних правил методами теорії графів. Такий підхід забезпечує достатній рівень формалізації для автоматичного виявлення порушень архітектури та створює основу для розширення системи новими правилами й механізмами аналізу. 2.3. Формалізація архітектурних правил і механізми їх перевірки Ефективність автоматизованої детекції архітектурних порушень у Python- проєктах визначається не лише здатністю системи коректно витягувати залежності з коду, а й наявністю формалізованих правил, які однозначно описують допустиму архітектуру. У межах системи Archcon архітектурні правила розглядаються як набір обмежень на взаємодію між елементами проєкту (пакетами, модулями, класами), що можуть бути перевірені на основі орієнтованого графа залежностей, побудованого з AST-фактів. Формалізація правил потрібна для того, щоб перетворити неявні архітектурні домовленості (наприклад,

Цитирования: **0,04%** id: **32**

«domain не має залежати від infrastructure»

) на перевірювані інваріанти. ( ) У загальному вигляді проєкт описується орієнтованим графом ⬚⬚ = ⬚⬚, ⬚⬚ , де ⬚⬚ — множина вершин (модулів або пакетів), а ⬚⬚ ⊆ ⬚⬚ × ⬚⬚ — множина орієнтованих ребер залежності. Кожна вершина може мати атрибути, наприклад ( ) ( ) належність до шару ⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚ ⬚⬚ , доменної підсистеми ⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚ ⬚⬚ або категорії ⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚/⬚⬚h⬚⬚⬚⬚⬚⬚ – ⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚. Архітектурні правила тоді задаються як предикати над вершинами й ребрами, що визначають дозволені або заборонені відношення. Такий підхід є універсальним і придатним для опису обмежень як шарових архітектур, так і Clean/Hexagonal стилів. Найпростішим класом правил є

Цитирования: **0,02%** id: **33**

“правила заборонених залежностей”

між множинами компонентів. Нехай ⬚⬚ ⊆ ⬚⬚ і ⬚⬚ ⊆ ⬚⬚ — множини вершин (наприклад, пакети домену та інфраструктури). Тоді правило

Цитирования: **0,03%** id: **34**

«A не залежить від B»

( ) формально задається як заборона ребер ⬚⬚ ∩ ⬚⬚ × ⬚⬚ =\⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚⬚

Цитирования: **0,01%** id: **36**

«зворотні»

залежності, коли нижчий шар імпортує вищий. У Clean Architecture та Hexagonal Architecture навпаки ключовим є обмеження на залежність внутрішніх шарів від зовнішніх, тому напрям дозволених залежностей визначається правилами інверсії залежностей. Окремий клас становлять

Цитирования: **0,01%** id: **37**

“правила циклічності”,

які забороняють наявність циклів у графі залежностей між певними компонентами. Формально це означає відсутність циклу   →   → ⋯ →   . На практиці такі порушення 0 1 0 виявляються шляхом пошуку сильно зв’язаних компонент   або через   із маркуванням вершин. У Python циклічні залежності часто призводять не лише до архітектурних проблем, а й до реальних помилок імпорту під час виконання, тому їх детекція є важливою функцією системи. Для складніших сценаріїв застосовуються @правила досяжності (reachability rules)”, які забороняють залежності не лише прямі, а й транзитивні. Наприклад, правило

Цитирования: **0,03%** id: **38**

«api не повинно бачити infrastructure»

означає заборону будь-якого шляху   \ *h* , навіть якщо прямого імпорту немає, але залежність виникає через проміжні модулі. Такі правила формалізуються через перевірку існування шляхів у графі, що реалізується пошуком у глибину/ширину або попереднім обчисленням транзитивного замикання для вибраних підграфів. З практичної точки зору важливо, щоб правила мали

Цитирования: **0,02%** id: **39**

“зручний формат опису”.

У системі Archcon доцільним є використання декларативного представлення (YAML/JSON), яке дозволяє задавати: (1) множини компонентів через шаблони шляхів/пакетів; (2) тип правила (forbid/allow/only); (3) рівень застосування (package/module/class); (4) додаткові параметри (транзитивність, виключення, винятки). Це забезпечує розширюваність і дає змогу адаптувати систему до різних архітектурних стилів без модифікації ядра алгоритмів. Механізм перевірки правил у загальному вигляді складається з трьох кроків. Спочатку правила інтерпретуються та перетворюються у множини вершин графа (за назвами пакетів, шляхами або регулярними виразами). Далі для кожного правила обирається відповідний алгоритм перевірки: пошук ребер між множинами, пошук циклів, перевірка напрямів залежностей за функцією шару, або пошук шляхів для транзитивних обмежень. На завершення результати агрегуються у перелік порушень із контекстом: правило, джерело залежності, цільова вершина, тип порушення, а також, за можливості, конкретні файли й рядки коду, які сформували відповідне ребро залежності. Отже, формалізація архітектурних правил у системі Archcon базується на графовій моделі залежностей Python-проєкту та поданні правил як предикатів і обмежень над ребрами й шляхами цього графа. Механізми перевірки реалізуються стандартними алгоритмами теорії графів і забезпечують можливість як простих заборон залежностей, так і більш складних перевірок шаровості, циклічності та транзитивних порушень. Такий підхід створює надійну основу для автоматизованого контролю архітектурної цілісності Python-проєктів та подальшого розширення набору правил. 2.4. Метрики оцінювання архітектурної якості та порушень Для обґрунтованого аналізу архітектурної якості програмного забезпечення недостатньо лише факту наявності або відсутності порушень архітектурних правил. Необхідним є використання кількісних метрик, які дозволяють оцінювати ступінь деградації архітектури, порівнювати різні версії системи та відстежувати динаміку змін у процесі розвитку проєкту. У межах системи Archcon метрики архітектурної якості визначаються на основі орієнтованого графа залежностей, побудованого з результатів аналізу абстрактних синтаксичних дерев Python-коду. Базовою групою є

Цитирования: **0,01%** id: **40**

“метрики залежностей”,

які характеризують структуру взаємодії між компонентами системи. До них належить кількість вершин і ребер графа, середній та максимальний ступінь входу й виходу вершин, а також щільність графа залежностей. Зростання кількості ребер і щільності графа, як правило, свідчить про зниження модульності та підвищення рівня зв’язаності між компонентами. Для Python-проєктів ці показники є особливо важливими через відсутність жорстких обмежень на імпорти між модулями. Окрему групу становлять

Цитирования: **0,01%** id: **41**

“метрики циклічності”,

які відображають наявність і складність циклічних залежностей. Основними показниками є кількість циклів у графі, кількість сильно зв’язаних компонент, їхній середній та максимальний розмір. Наявність великих сильно зв’язаних компонент свідчить про архітектурну деградацію та ускладнює рефакторинг і тестування системи. У Python циклічні залежності можуть також мати прямі негативні наслідки для виконання програми, що підсилює важливість цієї групи метрик. Для оцінювання дотримання шарової архітектури та принципу інверсії залежностей використовуються

Цитирования: **0,02%** id: **42**

“метрики порушень правил”.

До них належить загальна кількість архітектурних порушень, кількість порушень певного типу (заборонені залежності, зворотні залежності, транзитивні порушення), а також частка компонентів, залучених у порушення. Такі метрики дозволяють не лише фіксувати факт проблеми, а й оцінювати її масштаб і критичність у контексті всієї системи. Важливою є група

Цитирования: **0,02%** id: **43**

“метрик локалізації порушень”,

які показують, наскільки архітектурні дефекти зосереджені в окремих компонентах або розподілені по всій системі. Наприклад, можна визначати кількість порушень на пакет або модуль, середню кількість порушень на компонент та компоненти з найбільшим числом порушень. Висока концентрація проблем у певних частинах системи може свідчити про архітектурні

Цитирования: **0,01%** id: **44**

«вузькі місця»

або некоректно спроєктовані підсистеми. Для динамічного аналізу розвитку архітектури використовуються

Цитирования: **0,01%** id: **45**

“порівняльні метрики”,

які дозволяють оцінювати зміни архітектурної якості між різними версіями проєкту. До таких метрик належать різниця в кількості порушень, зміна щільності графа залежностей, зменшення або зростання кількості циклів. У поєднанні з історією комітів ці показники дають змогу оцінювати ефективність архітектурних рішень і рефакторингів у часі. З практичної точки зору доцільно вводити

Цитирования: **0,01%** id: **46**

“агреговані показники”,

які узагальнюють стан архітектури у вигляді інтегральної оцінки. Така оцінка може базуватися на зваженій сумі нормалізованих метрик, де вагові коефіцієнти визначаються важливістю відповідних типів порушень. Наприклад, порушення інверсії залежностей або наявність великих циклів можуть мати більшу вагу, ніж поодинокі заборонені імпорти. Подібний підхід дозволяє порівнювати різні проєкти або версії системи за єдиним показником архітектурної якості. Отже, метрики оцінювання архітектурної якості в системі Archcon формують кількісну основу для аналізу структури Python-проєктів і виявлених архітектурних порушень. Вони доповнюють механізми формалізованої перевірки правил, дозволяють об’єктивно оцінювати стан архітектури, відстежувати її еволюцію та обґрунтовувати необхідність архітектурних змін і рефакторингу. Оцінювання архітектурної якості Python-проєкту в системі Archcon ( ) здійснюється на основі орієнтованого графа залежностей ⬚⬚ = ⬚⬚, ⬚⬚ , побудованого в результаті аналізу абстрактних

синтаксичних дерев. Для кожної групи метрик використовується окрема методика визначення показників, що забезпечує відтворюваність результатів і можливість порівняння між різними проєктами або версіями системи. Метрики залежностей обчислюються безпосередньо зі структури графа ⟨⟩ . | | Кількість вершин ⟨⟩ визначається як загальна кількість унікальних компонентів вибраного рівня абстракції (пакети або модулі), які беруть участь у внутрішніх | | залежностях проєкту. Кількість ребер ⟨⟩ відповідає загальній кількості зафіксованих залежностей, зокрема імпортів між внутрішніми компонентами. − ( ) Ступінь входу вершини ⟨⟩ визначається як кількість ребер, що входять у вершину ⟨⟩ , і відображає кількість компонентів, які залежать від даного. + ( ) Ступінь виходу ⟨⟩ визначається як кількість ребер, що виходять з вершини ⟨⟩ , і характеризує рівень залежності компонента від інших. Середні значення ступенів входу та виходу обчислюються як середнє арифметичне для всіх вершин ( ) | | | | | | графа. Щільність графа визначається як відношення ⟨⟩ /(⟨⟩ · ⟨⟩ − 1) і використовується як узагальнений показник зв’язаності архітектури. Методика визначення метрик циклічності Для виявлення циклічних залежностей у графі використовується алгоритм пошуку сильно зв’язаних компонент ⟨⟩ . Кожна сильно зв’язана компонента з кількістю вершин більше однієї або з петлею виду ⟨⟩ → ⟨⟩ вважається циклічною. Кількість таких компонент використовується як показник кількості циклів у системі. Розмір циклу визначається кількістю вершин у відповідній сильно зв’язаній компоненті. Для оцінювання складності циклічних залежностей обчислюються середній і максимальний розміри сильно зв’язаних компонент. Наявність великих SCC інтерпретується як ознака високого рівня архітектурної зв’язаності та потенційної складності рефакторингу. Методика визначення метрик порушень архітектурних правил Метрики порушень базуються на результатах перевірки формалізованих архітектурних правил. Для кожного правила система Archcon формує множину знайдених порушень, де кожне порушення відповідає конкретному факту невідповідності графа заданому обмеженню. Загальна кількість архітектурних порушень визначається як сума всіх зафіксованих порушень для всіх правил. Додатково обчислюється кількість порушень за типами, наприклад: заборонені залежності, порушення напрямів шарів, транзитивні порушення або циклічні порушення. Для кожного типу може визначатися частка від загальної кількості порушень, що дозволяє оцінити домінуючі проблеми архітектури. Методика визначення метрик локалізації порушень Для аналізу розподілу архітектурних дефектів використовується агрегація порушень за компонентами. Для кожної вершини ⟨⟩ ∈ ⟨⟩ визначається кількість порушень, у яких вона виступає джерелом або цільовою вершиною забороненої залежності. На основі цих даних обчислюється середня кількість порушень на компонент та визначаються компоненти з максимальним числом порушень. Додатково може застосовуватися коефіцієнт концентрації порушень, який визначається як відношення кількості порушень у топ-(k) проблемних компонентах до загальної кількості порушень у системі. Високе значення цього показника свідчить про локалізований характер архітектурних проблем, тоді як низьке — про їх системний характер. Методика визначення порівняльних метрик Для оцінювання динаміки архітектурної якості використовується порівняння значень метрик для різних версій проєкту. Для кожної метрики ⟨⟩ визначається її значення у версіях ⟨⟩ і ⟨⟩ , після чого обчислюється абсолютна або 1 2 ( ) ( ) відносна зміна ⟨⟩ ⟨⟩ = ⟨⟩ ⟨⟩ − ⟨⟩ ⟨⟩ . 2 1 Аналіз знаку та величини ⟨⟩ ⟨⟩ дозволяє зробити висновки щодо покращення або погіршення архітектури. Наприклад, зменшення кількості порушень і циклів інтерпретується як позитивний результат архітектурних змін, тоді як зростання щільності графа або кількості заборонених залежностей — як негативна тенденція. Методика визначення агрегованої оцінки архітектурної якості Для отримання інтегральної оцінки архітектурної якості використовується нормалізація окремих метрик до інтервалу ([0,1]) та їх зважене агрегування. Кожній метриці ⟨⟩ призначається ваговий коефіцієнт ⟨⟩ , що відображає її ⟨⟩ ⟨⟩ важливість. Загальна оцінка архітектурної якості обчислюється як зважена сума нормалізованих значень метрик. Такий підхід дозволяє звести багатовимірну оцінку архітектури до одного показника, зберігаючи при цьому можливість детального аналізу за окремими метриками у разі потреби. РОЗДІЛ 3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ АВТОМАТИЗОВАНОЇ СИСТЕМИ ARCHCON 3.1. Архітектура та компоненти автоматизованої системи Archcon Автоматизована система Archcon спроєктована як модульна програмна система, призначена для статичного аналізу Python-проєктів і детекції архітектурних порушень на основі аналізу абстрактних синтаксичних дерев. Під час проєктування архітектури системи було враховано вимоги до розширюваності, адаптивності до різних

архітектурних стилів, а також можливість інтеграції з існуючими процесами розробки програмного забезпечення. Архітектура Archcon орієнтована на чітке розмежування відповідальностей між компонентами та відповідає принципам слабкої зв’язаності й високої когезії. Рис. 3.1 Узагальена модель системи Archcon У загальному вигляді Archcon складається з послідовності взаємопов’язаних компонентів, які реалізують повний цикл аналізу — від отримання вихідного коду до формування звіту про архітектурний стан системи. Центральним елементом є керуючий модуль, який координує виконання всіх етапів аналізу, ініціює обробку проєкту та передає результати між підсистемами. Саме цей компонент відповідає за керування життєвим циклом аналізу та взаємодію з користувачем або зовнішніми інструментами. Першим функціональним компонентом системи є модуль сканування проєкту, який відповідає за виявлення та підготовку вхідних даних. Він здійснює рекурсивний обхід файлової системи, відбирає Python-файли відповідно до заданих критеріїв, ігнорує службові каталоги та нормалізує шляхи до модулів. На цьому етапі формується уявлення про структуру проєкту, що є необхідною передумовою для коректного аналізу залежностей. Ключовим аналітичним компонентом системи є модуль побудови та обходу AST, який реалізує синтаксичний розбір Python-коду з використанням стандартних засобів мови. Цей модуль виконує побудову абстрактних синтаксичних дерев для кожного файлу та здійснює їх обхід із метою вилучення структурних фактів. У результаті формуються дані про імпорти, оголошення класів і функцій, наслідування та інші конструкції, що мають архітектурне значення. Зібрані структурні факти передаються до модуля побудови графа залежностей, який агрегує інформацію з усіх файлів і формує орієнтований граф залежностей проєкту. Цей компонент забезпечує перехід від локального аналізу окремих файлів до глобального уявлення про архітектуру системи. Граф залежностей є центральною моделлю, на основі якої виконуються всі подальші перевірки архітектурних правил і обчислення метрик. Для контролю відповідності реалізації задекларованій архітектурі використовується модуль перевірки архітектурних правил. Він інтерпретує правила, задані у декларативному вигляді, та застосовує їх до побудованого графа залежностей. У цьому модулі реалізуються алгоритми перевірки заборонених залежностей, контролю напрямів шарів, виявлення циклічних залежностей і транзитивних порушень. Результатом роботи модуля є перелік архітектурних порушень із зазначенням їх типу та залучених компонентів. Окремим компонентом системи є модуль обчислення метрик архітектурної якості, який використовує граф залежностей і результати перевірки правил для визначення кількісних показників стану архітектури. До таких показників належать метрики зв’язаності, циклічності, кількості та локалізації порушень. Цей модуль забезпечує можливість об’єктивного оцінювання архітектури та порівняння різних версій проєкту. Завершальним етапом роботи системи є модуль формування звітів, який відповідає за подання результатів аналізу у зручному для користувача вигляді. Він підтримує різні формати вихідних даних, зокрема текстові звіти для командного рядка та структуровані формати для інтеграції з системами безперервної інтеграції. Модуль звітності узагальнює інформацію про знайдені порушення та обчислені метрики, забезпечуючи наочність і практичну корисність результатів. Таким чином, архітектура автоматизованої системи Archcon являє собою послідовно організований набір спеціалізованих компонентів, кожен з яких відповідає за окремий аспект аналізу Python-коду. Такий підхід забезпечує гнучкість, розширюваність і можливість адаптації системи до різних архітектурних стилів і вимог, що є важливою передумовою її практичного використання у процесі розробки програмного забезпечення. Для уточнення моделі розробимо діаграми UML системи Archcon, а саме: 1) діаграма прецедентів (Рис. 3.2) 2) діаграма класів (Рис. 3.3); 3) діаграма послідовностей (Рис. 3.4); 4) діаграма розгортання (Рис. 3.5). Рис. 3.2 Діаграма прецедентів системи Archcon Діаграма прецедентів (рис. 3.2) відображає основні сценарії використання автоматизованої системи Archcon та взаємодію з нею зовнішніх акторів. На діаграмі виділено два ключові актори: розробник і система CI/CD. Розробник виступає основним користувачем системи, який здійснює налаштування архітектурних правил, ініціює аналіз Python-проєкту та переглядає результати у вигляді звітів. Система CI/CD використовує Archcon у межах автоматизованих конвеєрів збірки з метою контролю архітектурної якості на етапах безперервної інтеграції. Центральним прецедентом є запуск аналізу Python-проєкту, який включає перевірку архітектурних правил, обчислення метрик архітектурної якості та формування звіту. Окремо виділено прецедент інтеграції з CI/CD, який передбачає використання коду завершення та артефактів звіту для автоматичного прийняття рішень щодо якості коду. Таким чином, діаграма демонструє, що Archcon може використовуватися як у ручному режимі

розробником, так і в автоматизованих процесах розробки. Рис. 3.3. Діаграма класів системи Archon Діаграма класів (рис. 3.3) відображає узагальнену статичну структуру системи Archcon та основні програмні компоненти, з яких вона складається. Центральним класом є Orchestrator, який координує роботу всіх інших компонентів і реалізує основний сценарій аналізу проєкту. Клас ArchconCLI слугує точкою входу в систему та відповідає за приймання параметрів запуску з командного рядка. Окремі класи відповідають за спеціалізовані етапи аналізу: ProjectScanner здійснює пошук Python-файлів, ASTAnalyzer виконує аналіз абстрактних синтаксичних дерев, GraphBuilder формує орієнтований граф залежностей, RuleEngine перевіряє архітектурні правила, а MetricsEngine обчислює показники архітектурної якості. Клас Reporter відповідає за подання результатів у різних форматах, тоді як AnalysisResult слугує контейнером для агрегованих результатів аналізу. Ді Рис. 3.4 Діаграма послідовностей системи Archcon Діаграма послідовностей ілюструє динамічну взаємодію між компонентами системи Archcon під час виконання типового сценарію аналізу Python-проєкту. Процес починається з ініціації аналізу розробником через командний рядок. Далі керування передається центральному компоненту Orchestrator, який послідовно викликає інші модулі системи. Спочатку здійснюється завантаження конфігурації архітектурних правил, після чого виконується сканування проєкту та побудова абстрактних синтаксичних дерев. На наступному етапі зібрані структурні факти використовуються для побудови графа залежностей, до якого застосовуються архітектурні правила. Після виявлення порушень система обчислює метрики архітектурної якості та формує звіт. Діаграма розгортання (рис. 3.5) відображає фізичне розміщення компонентів системи Archcon та їх взаємодію з оточенням. Основним вузлом розгортання є робоча машина розробника або агент CI/CD, на якому встановлено Archcon у вигляді Python-пакета. На цьому ж вузлі розміщується вихідний код Python-проєкту, конфігураційні файли архітектурних правил та результати аналізу у вигляді звітів. Рис. 3.5. Діаграма розгортання системи Archon Окремо на рис. 3.5 показано взаємодію з репозиторієм керування версіями, з якого система CI/CD отримує вихідний код проєкту. Після виконання аналізу результати можуть зберігатися як артефакти конвеєра або використовуватися для автоматичного контролю якості збірки. Діаграма підкреслює, що Archcon не потребує складної інфраструктури та може бути легко інтегрована в існуючі середовища розробки. 3.2. Обґрунтування вибору програмних засобів і бібліотек реалізації Реалізація автоматизованої системи детекції архітектурних порушень Archcon вимагала добору програмних засобів і бібліотек, які б забезпечували коректний статичний аналіз Python-коду, гнучкість у формалізації архітектурних правил, достатню продуктивність та можливість інтеграції з існуючими процесами розробки програмного забезпечення. Вибір інструментарію здійснювався з урахуванням специфіки мови Python, вимог до розширюваності системи та доцільності використання стандартних і широко підтримуваних рішень. Як основну мову реалізації системи Archcon було обрано Python, оскільки він є об’єктом аналізу в межах дослідження і водночас надає потужні вбудовані засоби для роботи з власною синтаксичною моделлю. Використання Python дозволяє уникнути семантичних розбіжностей між мовою аналізу та мовою реалізації, спрощує обробку модульної структури проєктів і забезпечує природну інтеграцію з екосистемою інструментів статичного аналізу. Ключовим програмним засобом для аналізу вихідного коду є стандартний модуль `ast`, який входить до стандартної бібліотеки Python. Саме цей модуль забезпечує побудову абстрактного синтаксичного дерева, яке точно відображає структуру програми так, як її інтерпретує Python-інтерпретатор. Використання стандартного модуля `ast` є обґрунтованим з огляду на його стабільність, відсутність зовнішніх залежностей та відповідність актуальним версіям мови. Альтернативні підходи, зокрема аналіз байт-коду або сторонні парсери, були відхилені через складність інтерпретації результатів або надмірну залежність від реалізації конкретної версії інтерпретатора. Для реалізації обходу абстрактних синтаксичних дерев використано механізм `ast.NodeVisitor`, який надає зручний і формалізований спосіб обробки різних типів вузлів AST. Такий підхід дозволяє чітко розмежувати логіку аналізу імпортів, визначень класів і функцій, викликів та інших синтаксичних конструкцій, що є важливим для архітектурного аналізу. Використання патерну

Цитирования: **0,01%** id: **47**

«Visitor»

також сприяє розширюваності системи, оскільки дає змогу додавати нові типи перевірок без модифікації базового алгоритму обходу. ast.NodeVisitor — це базовий клас у модулі ast

стандартної бібліотеки Python, який реалізує патерн проєктування Відвідувач (Visitor). Він дозволяє відокремити алгоритм обходу дерева від самої структури об'єктів (вузлів дерева). Алгоритм диспетчеризації (Dispatch Mechanism). Коли запускаємо обхід дерева, викликається метод visit(node). Логіка його роботи наступна: Визначення типу вузла: Метод отримує ім'я класу вузла (наприклад, Import, ClassDef, FunctionDef). Формування імені методу: Система шукає у вашому класі метод з іменем visit_ + ІмяВузла (наприклад, visit_Import). Виклик обробника: Якщо метод існує (наприклад, ви його написали), він викликається. Якщо методу не існує, викликається метод generic_visit(node), який автоматично проходить по всіх дочірніх вузлах поточного вузла. Приклад використання ast у лістингах 3.1 та 3.2. Лістинг 3.1 import ast class DependencyVisitor(ast.NodeVisitor): def __init__(self): # Тут зберігатимемо знайдені залежності self.dependencies = [] def visit_Import(self, node): """ Обробляє конструкції типу: import os, sys """ for alias in node.names: self.dependencies.append(alias.name) # generic_visit не потрібен, бо в імпортах немає вкладеного коду def visit_ImportFrom(self, node): """ Обробляє конструкції типу: from datetime import datetime """ if node.module: self.dependencies.append(node.module) else: # Це обробка відносних імпортів (наприклад, from . import utils) self.dependencies.append(f

Цитирования: **0,01%** id: **48**

".level_{node.level}"

) # --- Емуляція роботи системи Archcon --- source_code = """ import os import json from app.domain import models from app.infrastructure.db import repository class UserController: def get_user(self): pass """ # 1. Створюємо дерево tree = ast.parse(source_code) # 2. Ініціалізуємо відвідувача visitor = DependencyVisitor() # 3. Запускаємо обхід visitor.visit(tree) print(

Цитирования: **0,01%** id: **49**

"Знайдені залежності:",

visitor.dependencies) # Результат: ['os', 'json', 'app.domain', 'app.infrastructure.db'] Листинг 3.1 є прототипом ядра Archcon. Він знаходить усі зовнішні залежності файлу. Лістинг 3.2 class ContextAwareVisitor(ast.NodeVisitor): def __init__(self): self.current_context =

Цитирования: **0,04%** id: **50**

"Module Level" self.violations = []

def visit_ClassDef(self, node): # Зберігаємо попередній контекст prev_context = self.current_context # Встановлюємо новий self.current_context = f

Цитирования: **0,02%** id: **51**

"Class: {node.name}"

! ВАЖЛИВО: Продовжуємо обхід всередину класу self.generic_visit(node) # Відновлюємо контекст при виході self.current_context = prev_context def visit_FunctionDef(self, node): prev_context = self.current_context self.current_context = f"Function: {node.name}" self.generic_visit(node) self.current_context = prev_context def visit_Import(self, node): # Якщо імпорт знайдено не на рівні модуля, це може бути цікаво if "Function" in self.current_context: for alias in node.names: print(f" ПОПЕРЕДЖЕННЯ: Локальний імпорт '{alias.name}' " f

Цитирования: **0,03%** id: **52**

"знайдено всередині '{self.current_context}'"

) # --- Тест --- code_with_smell = """ class DataProcessor: def process(self): import pandas as pd # Локальний імпорт! pass """ visitor = ContextAwareVisitor() visitor.visit(ast.parse(code_with_smell)) Для Archcon часто важливо знати не просто що імпортується, а де саме. Наприклад, імпорт усередині функції (local import) часто є

Цитирования: **0,01%** id: **53**

"запашком"

(code smell). Лістинг 3.2 показує, як відстежувати батьківський контекст. Для представлення архітектури проєкту у вигляді орієнтованого графа залежностей доцільним є використання бібліотек для роботи з графами. У межах реалізації Archcon може застосовуватися бібліотека NetworkX, яка надає широкий набір структур даних і алгоритмів теорії графів, зокрема для пошуку циклів, сильно зв’язаних компонент, досяжності та аналізу шляхів. Вибір NetworkX обґрунтований її зрілістю, хорошою документацією та відповідністю науково-дослідницьким завданням. Альтернативою є власна реалізація

графових структур, однак це ускладнило б розробку та знизило надійність алгоритмів. Для опису архітектурних правил і параметрів аналізу використано декларативні формати конфігурації, зокрема YAML або JSON. Таке рішення дозволяє відокремити логіку перевірки від конкретних архітектурних вимог проєкту, забезпечує зручність редагування правил і полегшує інтеграцію з системами CI/CD. Використання стандартних бібліотек Python для роботи з цими форматами зменшує кількість зовнішніх залежностей і підвищує переносимість системи. Для взаємодії з користувачем і запуску аналізу обрано інтерфейс командного рядка (CLI), реалізований із використанням стандартних засобів або бібліотек типу `argparse` чи `click`. CLI-інтерфейс є універсальним рішенням, яке дозволяє використовувати Archcon як локально розробником, так і в автоматизованих середовищах безперервної інтеграції. Такий підхід також відповідає практиці використання інструментів статичного аналізу в сучасних процесах розробки. Для формування результатів аналізу застосовуються стандартні засоби серіалізації даних і генерації звітів у текстовому та структурованому форматах. Це забезпечує можливість подальшої обробки результатів іншими інструментами, а також спрощує порівняння різних запусків аналізу під час експериментального дослідження. Отже, вибір програмних засобів і бібліотек для реалізації системи Archcon базується на використанні стандартних і широко підтримуваних компонентів екосистеми Python, що забезпечує коректність аналізу, розширюваність системи та зручність її практичного застосування. Обраний інструментарій є достатнім для реалізації поставлених завдань і водночас не створює надмірної складності або залежності від специфічних зовнішніх рішень. 3.3. Реалізація ключових класів та логіки системи Archcon Реалізація системи Archcon була побудована за модульним принципом, де кожен ключовий етап аналізу Python-проєкту відокремлено у відповідний клас або підсистему. Такий підхід забезпечив розмежування відповідальностей між компонентами, спростив тестування та створив основу для розширення системи новими правилами та метриками без суттєвих змін у вже реалізованому коді. Центральною ідеєю реалізації є конвеєрний процес: сканування проєкту → побудова AST → вилучення фактів → побудова графа → перевірка правил → обчислення метрик → формування звіту. Керування життєвим циклом аналізу реалізовано у класі Orchestrator (дивись додаток А), який виступає координатором виконання. Він приймає шлях до проєкту та конфігурації правил, ініціює роботу сканера, викликає аналізатор AST, передає зібрані факти до модуля побудови графа, запускає механізм перевірки правил і метрик та завершує процес формуванням звіту. Саме Orchestrator забезпечує цілісність процесу та виступає єдиною точкою інтеграції для командного інтерфейсу і середовищ CI/CD. У реалізації важливо, що Orchestrator не містить логіки аналізу як такої, а лише виконує послідовні виклики підсистем і агрегує результати, що відповідає принципу єдиної відповідальності. Компонент ProjectScanner (дивись додаток Б) реалізує підготовчий етап — формування множини Python-файлів, які підлягають аналізу. Його логіка включає рекурсивний обхід файлової структури проєкту, фільтрацію службових директорій та нормалізацію шляхів до модулів. На цьому етапі також формується відображення

Цитирования: **0,03%** id: **54**

«файл → модульне ім’я»,

необхідне для коректного зіставлення залежностей у подальших етапах. У контексті Python це критично, оскільки одна й та сама залежність може бути виражена як абсолютним імпортом, так і відносним, тому нормалізація модульних імен є передумовою точного аналізу. Фрагмент ProjectScanner представлено у лістингу 3.3 який здійснює відбір релевантних файлів + контроль шуму (venv/build/cache). Лістинг 3.3 from pathlib import Path import fnmatch class ProjectScanner: DEFAULT_EXCLUDE_DIRS = {

Цитирования: **0,01%** id: **55**

".git",

Цитирования: **0,01%** id: **56**

"__pycache__",

Цитирования: **0,01%** id: **57**

".venv",

Цитирования: **0,01%** id: **58**

"venv",

Цитирования: **0,01%** id: **59**

"dist",

Цитирования: **0,01%** id: **60**

"build"

} def scan(self, project_path: str, config: dict | None = None) - list[str]: root = Path(project_path).resolve() cfg = config or {} exclude_dirs = set(cfg.get(

Цитирования: **0,01%** id: **61**

"exclude_dirs", [])

)) | self.DEFAULT_EXCLUDE_DIRS exclude_globs = cfg.get("exclude_globs", []) files: list[str] = [] for p in root.glob("**/*.py"): if any(d in p.parts for d in exclude_dirs): continue if any(fnmatch.fnmatch(p.as_posix(), g) for g in exclude_globs): continue files.append(str(p)) return sorted(files) Аналіз вихідного коду на основі абстрактних синтаксичних дерев реалізовано в класі ASTAnalyzer (дивись додаток Б), який поєднує парсинг файлів та обхід AST. Для кожного файлу виконується синтаксичний розбір з використанням стандартного модуля `ast`, після чого дерево обходиться за допомогою механізму Visitor. У процесі обходу формуються структурні факти, що мають архітектурне значення, насамперед залежності типу `import` та `from ... import ...`. За потреби додатково можуть реєструватися факти про успадкування, виклики, використання атрибутів, а також місця виникнення залежності (файл і позиція в коді). Якщо під час парсингу виникає синтаксична помилка, файл пропускається з фіксацією попередження, що забезпечує стійкість системи до частково некоректного коду або проміжних станів репозиторію. У лістингу 3.4 показано парсинг файлу в AST + запуск Visitor. Лістинг 3.4 import ast from pathlib import Path class ASTAnalyzer: def analyze(self, files: list[str], project_root: str): store = FactsStore(project_root=project_root) for fp in files: code = Path(fp).read_text(encoding="utf-8

Цитирования: **0,01%** id: **62**

", errors="

replace

Цитирования: **0,1%** id: **63**

") try: tree = ast.parse(code, filename=fp) except SyntaxError as e: store.add_parse_error(fp, f"

SyntaxError: {e.msg} (line {e.lineno})") continue src_mod = module_name_from_path(project_root, fp) _ImportVisitor(src_mod, fp, store).visit(tree) return store Для збереження результатів аналізу AST реалізовано структуру Facts та окремий компонент Facts Store (додаток Б), який акумулює отримані факти в уніфікованому вигляді. Це забезпечує відокремлення процесу збору інформації від процесу її використання. Наприклад, модуль побудови графа залежностей працює не з AST напряму, а з набором фактів, де залежності подані у вигляді пар

Цитирования: **0,02%** id: **64**

«джерело → ціль»

із нормалізованими іменами модулів або пакетів. Такий проміжний рівень абстракції підвищує стабільність і повторне використання результатів, а також дозволяє легко додавати нові типи фактів без зміни основних алгоритмів. Лістинг 3.5 демонструє витяг залежностей з Import / ImportFrom. visit_Import фіксує залежність src - alias.name. visit_ImportFrom додатково враховує level (відносні імпорти) і викликає нормалізацію resolve_from_import(). Лістинг 3.5 import ast class _ImportVisitor(ast.NodeVisitor): def __init__(self, src_module: str, src_file: str, store: "FactsStore

Цитирования: **0,24%** id: **65**

"): self.src_module = src_module self.src_file = src_file self.store = store def visit_Import(self, node: ast.Import): for alias in node.names: self.store.add_import(src=self.src_module, dst=alias.name, file=self.src_file, line=getattr(node, "

lineno", 0),) self.generic_visit(node) def visit_ImportFrom(self, node: ast.ImportFrom): base = node.module or "" level = getattr(node,

Цитирования: **0,01%** id: **66**

"level",

0) or 0 dst = resolve_from_import(self.src_module, base, level) if dst: self.store.add_import(src=self.src_module, dst=dst, file=self.src_file, line=getattr(node,

Цитирования: **0,01%** id: **67**

"lineno",

0),) self.generic_visit(node) Лістинг 3.6 показує як реалізується мінімальне сховище фактів, відокремлює

Цитирования: **0,01%** id: **68**

“збір”

від

Цитирования: **0,01%** id: **69**

“перевірки/графа”.

Лістинг 3.6 from dataclasses import dataclass, field @dataclass class FactsStore: project_root: str imports: list[dict] = field(default_factory=list) parse_errors: list[dict] = field(default_factory=list) def add_import(self, src: str, dst: str, file: str, line: int): self.imports.append({

Цитирования: **0,01%** id: **70**

"src":

src,

Цитирования: **0,01%** id: **71**

"dst":

dst,

Цитирования: **0,01%** id: **72**

"file":

file,

Цитирования: **0,01%** id: **73**

"line":

line}) def add_parse_error(self, file: str, message: str): self.parse_errors.append({

Цитирования: **0,01%** id: **74**

"file":

file,

Цитирования: **0,01%** id: **75**

"message":

message}) Побудова орієнтованого графа залежностей реалізується у класі GraphBuilder (додаток Б). Він агрегує факти з усіх файлів, формує множину вершин і ребер та виконує класифікацію залежностей на внутрішні й зовнішні. На практиці це означає, що з графа можуть вилучатися або маркуватися залежності на сторонні бібліотеки, оскільки архітектурні правила найчастіше застосовуються до внутрішньої структури проєкту. GraphBuilder також підтримує згортання графа на різних рівнях абстракції, наприклад, до рівня пакетів, що є важливим для перевірки шарової архітектури та Clean/Hexagonal-підходів. Лістинг 3.7 містить опис побудови орієнтованого графа залежностей. Лістинг 3.7 from collections import defaultdict class GraphBuilder: def build(self, facts: FactsStore, config: dict | None = None): edges = set() edge_sources = defaultdict(list) for imp in facts.imports: u, v = imp[

Цитирования: **0,03%** id: **76**

"src"], imp["dst"]

if u != v: edges.add((u, v)) edge_sources[(u, v)].append((imp[

Цитирования: **0,03%** id: **77**

"file"], imp["line"]

)) return {

Цитирования: **0,01%** id: **78**

"edges":

edges,

Цитирования: **0,01%** id: **79**

"sources":

edge_sources} Компонент RuleEngine (додаток Б) реалізує механізми формалізованої перевірки архітектурних правил. Правила завантажуються з конфігурації та інтерпретуються як обмеження над множинами вершин та ребер графа. Залежно від типу правила застосовується відповідний алгоритм: для заборони залежностей перевіряється наявність ребер між заданими множинами; для правил шаровості контролюється напрям залежностей між шарами; для циклічних порушень виконується пошук сильно зв’язаних компонент; для транзитивних порушень здійснюється перевірка досяжності. Результатом роботи RuleEngine є перелік порушень із зазначенням типу, залучених компонентів і, за можливості, місця походження залежності в коді. Важливо, що RuleEngine не змінює граф і не

Цитирования: **0,01%** id: **80**

«виправляє»

порушення, а лише забезпечує їхню детекцію та опис. На основі побудованого графа та знайдених порушень компонент MetricsEngine обчислює метрики архітектурної якості. До них належать метрики зв’язаності (кількість вершин і ребер, ступені входу/виходу, щільність), метрики циклічності (кількість SCC і їхній розмір), а також метрики порушень (загальна кількість, розподіл за типами, локалізація по модулях/пакетах). Для підтримки експериментального дослідження MetricsEngine формує показники, що можуть бути використані для порівняння стану архітектури

Цитирования: **0,01%** id: **81**

«до»

і

Цитирования: **0,01%** id: **82**

«після»

рефакторингу або впровадження правил. Завершальним компонентом є Reporte, який перетворює результати аналізу на зручний для користувача формат. Reporter підтримує щонайменше текстове подання для командного рядка та структурований формат (наприклад JSON) для інтеграції із зовнішніми системами. У звіті відображається перелік архітектурних порушень із контекстом, а також узагальнені метрики. Для використання в CI/CD додатково формується код завершення, що дозволяє автоматично

Цитирования: **0,01%** id: **83**

«провалювати»

збірку при наявності критичних порушень або перевищенні заданих порогів метрик. Таким чином, реалізація ключових класів і логіки системи Archcon забезпечує повний цикл автоматизованого аналізу архітектурної якості Python- проєктів. Конвеєрна організація, наявність проміжного шару у вигляді структурних фактів, використання графової моделі залежностей і декларативних правил дозволяють системі бути розширюваною, відтворюваною та придатною для інтеграції в реальні процеси розробки. 3.4. Інтерфейс користувача та інтеграція системи у процес розробки Автоматизована система Archcon орієнтована на використання в реальних процесах розробки програмного забезпечення, тому під час її проєктування особлива увага приділялася зручності взаємодії з користувачем і можливості інтеграції з типовими інструментами життєвого циклу ПЗ. З огляду на характер задачі статичного аналізу та цільову аудиторію (розробники і команди розробки), основним інтерфейсом взаємодії з системою було обрано інтерфейс командного рядка, який забезпечує універсальність, автоматизованість і незалежність від конкретного середовища виконання. Інтерфейс командного рядка Archcon надає користувачу можливість запуску аналізу з мінімально необхідним набором параметрів, зокрема шляхом указання каталогу з Python-проєктом та файлу конфігурації архітектурних правил.

Додаткові параметри дозволяють керувати форматом звіту, місцем його збереження та політикою обробки виявлених порушень. Такий підхід відповідає усталеній практиці використання інструментів статичного аналізу та робить систему придатною як для інтерактивного використання розробником, так і для автоматизованих запусків. Важливим елементом інтерфейсу є структурований формат конфігурації архітектурних правил. Правила задаються у вигляді декларативного файлу, що дозволяє відокремити логіку аналізу від конкретних архітектурних вимог проєкту. Користувач може змінювати правила без модифікації коду системи, що суттєво підвищує гнучкість застосування Archcon у різних проєктах. Такий підхід також спрощує адаптацію системи до різних архітектурних стилів, зокрема шарової архітектури, Clean Architecture або Hexagonal Architecture. Результати аналізу подаються у вигляді звіту, який може бути згенерований у текстовому або структурованому форматі. Текстовий звіт орієнтований на швидкий перегляд у командному рядку та містить узагальнену інформацію про стан архітектури, ключові метрики та перелік виявлених порушень. Структурований формат призначений для подальшої машинної обробки, зокрема для інтеграції з зовнішніми системами моніторингу якості або для накопичення статистики у процесі експериментального дослідження. Інтеграція Archcon у процес розробки програмного забезпечення передбачає використання системи на різних етапах життєвого циклу. На етапі локальної розробки Archcon може запускатися розробником вручну для перевірки дотримання архітектурних правил перед фіксацією змін у репозиторії. У такому сценарії система виступає інструментом попереднього контролю архітектурної якості та сприяє ранньому виявленню архітектурних дефектів. На етапі безперервної інтеграції Archcon може бути інтегрована до конвеєрів CI/CD як окремий крок перевірки якості. Завдяки використанню коду завершення система здатна сигналізувати про наявність критичних архітектурних порушень, що дозволяє автоматично блокувати злиття змін або зупиняти збірку. Такий підхід забезпечує контроль архітектурної цілісності проєкту в динаміці та запобігає поступовій деградації архітектури. Окрему роль відіграє можливість збереження результатів аналізу у вигляді артефактів збірки. Це дозволяє відстежувати зміну архітектурних метрик у часі, порівнювати різні версії системи та використовувати результати аналізу для подальших досліджень або ухвалення архітектурних рішень. У поєднанні з декларативними правилами це створює передумови для впровадження практик архітектурного управління в командній розробці. Таким чином, інтерфейс користувача та механізми інтеграції системи Archcon були спроєктовані з урахуванням реальних потреб процесу розробки програмного забезпечення. Використання інтерфейсу командного рядка, декларативної конфігурації правил і можливостей інтеграції з CI/CD дозволяє застосовувати систему як інструмент постійного контролю архітектурної якості, що підвищує надійність і підтримуваність Python-проєктів. ВИСНОВКИ У магістерській роботі було розглянуто та вирішено актуальну науково-практичну задачу автоматизованої детекції архітектурних порушень у Python- проєктах на основі аналізу абстрактних синтаксичних дерев. Актуальність дослідження зумовлена зростанням складності програмних систем, широким використанням мови Python у проєктах різного масштабу та відсутністю вбудованих механізмів контролю архітектурних обмежень у цій мові. У таких умовах архітектурні порушення часто виявляються на пізніх етапах життєвого циклу програмного забезпечення, що ускладнює супровід, тестування та розвиток системи. У ході роботи було виконано аналіз сучасних архітектурних патернів програмного забезпечення та типових порушень їх реалізації, зокрема в контексті особливостей мови Python. Показано, що динамічна природа Python, відсутність жорстких обмежень на імпорти між модулями та широкі можливості метапрограмування істотно ускладнюють статичний аналіз архітектури та потребують спеціалізованих підходів. Проведений огляд існуючих інструментів статичного аналізу та контролю якості Python-коду засвідчив, що більшість із них орієнтовані переважно на стиль, синтаксичні помилки або локальні дефекти коду і не забезпечують повноцінного контролю архітектурних інваріантів. У роботі обґрунтовано доцільність використання абстрактних синтаксичних дерев як базової моделі для аналізу Python-коду. Показано, що AST дозволяє отримати структурне подання програми, незалежне від форматування та стилістичних особливостей, і є адекватною основою для вилучення архітектурно значущих залежностей. Запропоновано підхід до представлення Python-проєкту у вигляді орієнтованого графа залежностей, який формалізує архітектуру системи та дозволяє застосовувати алгоритми теорії графів для перевірки архітектурних правил. У межах дослідження було розроблено автоматизовану систему Archcon, архітектура якої базується на модульному та конвеєрному підходах. Реалізовано ключові компоненти системи: сканер проєкту,

аналізатор абстрактних синтаксичних дерев із використанням патерну Visitor, сховище структурних фактів, модуль побудови графа залежностей, механізм формалізованої перевірки архітектурних правил, підсистему обчислення метрик архітектурної якості та модуль формування звітів. Така організація забезпечує чітке розмежування відповідальностей, розширюваність системи та можливість її адаптації до різних архітектурних стилів. Важливим результатом роботи є формалізація архітектурних правил у вигляді обмежень над орієнтованим графом залежностей. Запропоновано механізми перевірки заборонених залежностей, шарової архітектури, циклічних і транзитивних порушень. На основі графової моделі визначено набір метрик, що дозволяють кількісно оцінювати архітектурну якість системи, рівень зв’язаності компонентів, наявність циклів і концентрацію архітектурних дефектів. Розроблено методики обчислення цих метрик, що забезпечують відтворюваність результатів та можливість порівняння різних версій програмного забезпечення. Окрему увагу приділено питанням інтерфейсу користувача та інтеграції системи Archcon у процес розробки програмного забезпечення. Обґрунтовано вибір інтерфейсу командного рядка як універсального засобу взаємодії, що дозволяє використовувати систему як у локальній розробці, так і в середовищах безперервної інтеграції. Показано, що використання декларативної конфігурації правил і механізмів коду завершення дає змогу ефективно інтегрувати Archcon у CI/CD-конвеєри та здійснювати постійний контроль архітектурної якості проєктів. Отримані результати мають як наукову, так і практичну цінність. Наукова новизна роботи полягає у застосуванні AST-орієнтованого підходу до формалізованого контролю архітектури Python-проєктів та у поєднанні цього підходу з графовою моделлю залежностей і системою метрик архітектурної якості. Практична цінність полягає у можливості використання розробленої системи Archcon як інструмента підтримки архітектурної дисципліни у реальних проєктах, а також як основи для подальших досліджень і розширень у галузі статичного аналізу програмного забезпечення. Подальші напрями розвитку роботи можуть включати розширення набору аналізованих фактів (наприклад, викликів методів і механізмів ін’єкції залежностей), підтримку типових анотацій Python, удосконалення мови опису архітектурних правил, а також інтеграцію з іншими інструментами аналізу якості коду. Це підтверджує, що запропонований підхід є перспективним і має потенціал для подальшого розвитку та практичного впровадження. СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 1) Буч Г. Об’єктно-орієнтований аналіз і проєктування з прикладами застосувань / пер. з англ. — Київ : Вільямс, 2003. — 720 с. 2) Коваль Г. О., Мельник С. М. Архітектура програмного забезпечення: принципи та підходи до проєктування : навч. посіб. — Львів : Видавництво Львівської політехніки, 2019. — 312 с. 3) Морозов О. М. Інженерія програмного забезпечення : підручник. — Київ : Академія, 2018. — 448 с. 4) Семеріков С. О., Теплицький І. О. Методи та засоби забезпечення якості програмного забезпечення : навч. посіб. — Кривий Ріг : Вид-во КДПУ, 2017. — 296 с. 5) Баранов В. О., Дорошенко А. Є. Статичний аналіз програм: теорія і практика // Вісник Національного технічного університету України

Цитирования: **0,01%** id: **84**

«КПІ».

Серія : Інформатика, управління та обчислювальна техніка. — 2016. — № 63. — С. 45–53. 6) ДСТУ ISO/IEC 25010:2015. Системна та програмна інженерія. Моделі якості систем та програмного забезпечення. — Київ : ДП

Цитирования: **0,01%** id: **85**

«УкрНДНЦ»,

2016. — 34 с. 7) ДСТУ ISO/IEC 12207:2016. Системна та програмна інженерія. Процеси життєвого циклу програмного забезпечення. — Київ : ДП

Цитирования: **0,01%** id: **86**

«УкрНДНЦ»,

2017. — 138 с. 8) Харченко В. С., Скляр В. В. Надійність та якість програмних систем : навч. посіб. — Харків : ХНУРЕ, 2015. — 260 с. 9) Кузьмінський А. М. Методологія наукових досліджень у галузі інформаційних технологій : навч. посіб. — Київ : КНТ, 2020. — 292 с. 10) Дорошенко А. Є., Шевченко В. О. Методи аналізу та верифікації програмних систем // Проблеми програмування. — 2018. — № 2–3. — С. 3–14. 11) Fowler M.

Цитирования: **0,03%** id: **87**

“Patterns of Enterprise Application Architecture”.

— Boston : Addison-Wesley, 2002. — 533 p. 12) Martin R. C.

Цитирования: **0,07%** id: **88**

“Clean Architecture: A Craftsman’s Guide to Software Structure and Design”.

— Boston : Prentice Hall, 2017. — 432 p. 13) Martin R. C.

Цитирования: **0,05%** id: **89**

“Agile Software Development: Principles, Patterns, and Practices”.

— Upper Saddle River : Prentice Hall, 2003. — 552 p. 14) Gamma E., Helm R., Johnson R., Vlissides J.

Цитирования: **0,05%** id: **90**

“Design Patterns: Elements of Reusable Object-Oriented Software”.

— Boston : Addison-Wesley, 1994. — 395 p. 15) Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.

Цитирования: **0,07%** id: **91**

“Pattern- Oriented Software Architecture. Volume 1: A System of Patterns”.

— Chichester : Wiley, 1996. — 476 p. 16) Evans E.

Цитирования: **0,06%** id: **92**

“Domain-Driven Design: Tackling Complexity in the Heart of Software”.

— Boston : Addison-Wesley, 2003. — 560 p. 17) Cockburn A. Hexagonal Architecture [Електронний ресурс]. — Режим доступу: [https://alistair.cockburn.us/hexagonal- architecture/](https://alistair.cockburn.us/hexagonal-architecture/). 18) Lattner C., Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation //

Цитирования: **0,07%** id: **93**

“Proceedings of the International Symposium on Code Generation and Optimization”.

— 2004. — P. 75–86. 19) Baxter I. D., Pidgeon C. Software Change through Design Maintenance //

Цитирования: **0,05%** id: **94**

“Proceedings of the International Conference on Software Maintenance”.

— 1997. — P. 250–259. 20) Python Software Foundation. The Python Language Reference [

Обнаружен Плагиат: **0,17%** https://web-crawler.plagiarism-detect... + 7 ресурсов! id: **95**

Електронний ресурс]. — Режим доступу: https://docs.python.org/3/reference/. 21) Python Software Foundation. ast — Abstract Syntax Trees [Електронний ресурс]. — Режим доступу:

[https://docs.python.org/3/library/ast.html](https://docs.python.org/3/library/ ast.html). 22) McCabe T. J. A Complexity Measure //

Цитирования: **0,03%** id: **96**

“IEEE Transactions on Software Engineering”.

— 1976. — Vol. SE-2, No. 4. — P. 308–320. 23) Mens T., Tourwé T. A Survey of Software Refactoring //

Цитирования: **0,03%** id: **97**

“IEEE Transactions on Software Engineering”.

— 2004. — Vol. 30, No. 2. — P. 126–139. 24) NetworkX Developers. NetworkX Documentation [

Обнаружен Плагиат: **0,35%** https://web-crawler.plagiarism-detect... + 7 ресурсов! id: **98**

Електронний ресурс]. — Режим доступу: [https://networkx.org/documentation/stable/](https://networkx.org/document ation/stable/). 25) Pylint Documentation [Електронний ресурс]. — Режим доступу: https://pylint.pycqa.org/. 26) Flake8: Your Tool for Style Guide Enforcement [Електронний ресурс]. — Режим доступу: https://flake8.pycqa.org/. 27) SonarSource. SonarQube Documentation [Електронний ресурс]. — Режим доступу:

https://docs.sonarqube.org/. 28) Newman S.

Цитирования: **0,03%** id: **99**

“Building Microservices: Designing Fine-Grained Systems”.

— Sebastopol : O’Reilly Media, 2015. — 280 p. 29) Systä T. Static Analysis Frameworks for Software Architecture Recovery //

Цитирования: **0,01%** id: **100**

“Software Architecture”.

— Berlin : Springer, 2004. — P. 173–185. 30) Pressman R. S., Maxim B. R.

Цитирования: **0,03%** id: **101**

“Software Engineering: A Practitioner’s Approach”.

— 8th ed. — New York : McGraw-Hill, 2014. — 976 p. ДОДАТКИ Додаток А. Код класу Orchestrator from __future__ import annotations from dataclasses import dataclass from enum import IntEnum from typing import Any, Dict, List, Optional, Protocol, Tuple # ========== # Domain models (мінімальні) # ========== @dataclass(frozen=True) class AnalysisResult: graph: Any violations: List[Dict[str, Any]] metrics: Dict[str, Any] report: Optional[str] = None class ExitCode(IntEnum): OK = 0 VIOLATIONS_FOUND = 1 ERROR = 2 # ========== # Протоколи компонентів # ========== class RulesConfig(Protocol): def load(self, path: str) - Dict[str, Any]: ... class ProjectScanner(Protocol): def scan(self, project_path: str, config: Optional[Dict[str, Any]] = None) - List[str]: ... class ASTAnalyzer(Protocol): def analyze(self, files: List[str], project_root: str) - Any: ... class GraphBuilder(Protocol): def build(self, facts: Any, config: Optional[Dict[str, Any]] = None) - Any: ... class RuleEngine(Protocol): def check(self, graph: Any, rules: Dict[str, Any], facts: Optional[Any] = None) - List[Dict[str, Any]]: ... class MetricsEngine(Protocol): def compute(self, graph: Any, violations: List[Dict[str, Any]], config: Optional[Dict[str, Any]] = None) - Dict[str, Any]: ... class Reporter(Protocol): def render(self, result: AnalysisResult, format: str =

Цитирования: **0,01%** id: **102**

"text",

output_path: Optional[str] = None, config: Optional[Dict[str, Any]] = None,) - str: ... # ========== # Orchestrator # ========== class Orchestrator: """ Orchestrator координує весь конвеєр аналізу: config - scan - AST - facts - graph - rules - metrics - report Принцип: Orchestrator НЕ містить логіки аналізу, лише керує викликами компонентів і агрегує результат. """ def __init__(self, rules_config: RulesConfig, scanner: ProjectScanner, ast_analyzer: ASTAnalyzer, graph_builder: GraphBuilder, rule_engine: RuleEngine, metrics_engine: MetricsEngine, reporter: Reporter,) - None: self._rules_config = rules_config self._scanner = scanner self._ast_analyzer = ast_analyzer self._graph_builder = graph_builder self._rule_engine = rule_engine self._metrics_engine = metrics_engine self._reporter = reporter def run(self, project_path: str, config_path: str, *, report_format: str =

Цитирования: **0,01%** id: **103**

"text",

report_output_path: Optional[str] = None, fail_on_violations: bool = True,) - Tuple[ExitCode, AnalysisResult]: """ Повертає (exit_code, AnalysisResult). exit_code: 0 - OK (порушень нема або fail_on_violations=False) 1 - є порушення (і fail_on_violations=True) 2 - помилка виконання (виняток/критична помилка) """ try: # 1) Load rules/config config = self._rules_config.load(config_path) # 2) Scan project - list of .py files files = self._scanner.scan(project_path, config=config) # 3) Build AST facts facts = self._ast_analyzer.analyze(files, project_root=project_path) # 4) Build dependency graph graph = self._graph_builder.build(facts, config=config) # 5) Check rules - violations rules = config.get(

Цитирования: **0,01%** id: **104**

"rules",

config) # дозволяє як

Цитирования: **0,01%** id: **105**

"rules: ...",

так і плоский формат violations = self._rule_engine.check(graph, rules=rules, facts=facts) # 6) Compute metrics metrics = self._metrics_engine.compute(graph, violations, config=config) # 7) Aggregate result result = AnalysisResult(graph=graph, violations=violations, metrics=metrics, report=None,) # 8) Render report (optionally write to file inside reporter) rendered = self._reporter.render(result, format=report_format, output_path=report_output_path, config=config,) result = AnalysisResult(graph=result.graph, violations=result.violations, metrics=result.metrics, report=rendered,) # 9) Exit code policy has_violations = len(violations) 0 if has_violations and fail_on_violations: return ExitCode.VIOLATIONS_FOUND, result return ExitCode.OK, result except Exception as exc: # Мінімальна діагностика (краще логувати у logger, але тут узагальнено) error_result = AnalysisResult(graph=None, violations=[{

Цитирования: **0,01%** id: **106**

"type":

Цитирования: **0,01%** id: **107**

"INTERNAL_ERROR",

Цитирования: **0,01%** id: **108**

"message":

str(exc),

Цитирования: **0,01%** id: **109**

"component":

Цитирования: **0,01%** id: **110**

"Orchestrator",

}], metrics={

Цитирования: **0,01%** id: **111**

"status":

Цитирования: **0,01%** id: **112**

"error"

}, report=None,) return ExitCode.ERROR, error_result # ========================== # Приклад використання (CLI) # ========================== # exit_code, result = orchestrator.run(# project_path=

Цитирования: **0,01%** id: **113**

"path/to/project",

config_path=

Цитирования: **0,01%** id: **114**

"rules.yml",

report_format=

Цитирования: **0,01%** id: **115**

"json",

report_output_path=

Цитирования: **0,01%** id: **116**

"archcon-report.json",

fail_on_violations=True, #) # print(result.report) # raise SystemExit(int(exit_code)) Додаток Б Код компонентів ProjectScanner, ASTAnalyzer, Facts Store, GraphBuilder, RuleEngine, MetricsEngine, Reporter from __future__ import annotations import ast import fnmatch import json import os from collections import Counter, defaultdict, deque from dataclasses import dataclass, field from pathlib import Path from typing import Any, Dict, Iterable, List, Optional, Sequence, Set, Tuple # ========================== # Facts model + Facts Store # ========================== @dataclass(frozen=True) class ImportFact:

Цитирования: **0,04%** id: **117**

"""Описує залежність імпорту: src_module - dst_module."""

src_module: str dst_module: str src_file: str lineno: int col_offset: int kind: str #

Цитирования: **0,01%** id: **118**

"import"

|

Цитирования: **0,01%** id: **119**

"from"

@dataclass class FactsStore: """ Сховище структурних фактів (мінімально необхідне для Archcon). За потреби легко розширити: calls, inheritance, defs, etc. """ project_root: str imports: List[ImportFact] = field(default_factory=list) parse_errors: List[Dict[str, Any]] = field(default_factory=list) def add_import(self, fact: ImportFact) - None: self.imports.append(fact) def add_parse_error(self, file_path: str, message: str) - None: self.parse_errors.append({

Цитирования: **0,01%** id: **120**

"file":

file_path,

Цитирования: **0,01%** id: **121**

"message":

message}) def as_dict(self) - Dict[str, Any]: return {

Цитирования: **0,01%** id: **122**

"project_root":

self.project_root,

Цитирования: **0,01%** id: **123**

"imports":

[fact.__dict__ for fact in self.imports],

Цитирования: **0,01%** id: **124**

"parse_errors":

list(self.parse_errors), } # ========== # ProjectScanner # ========== class ProjectScanner: """ Знаходить Python-файли в проєкті, застосовує exclude/ignore. Повертає список шляхів файлів (str), відносно/абсолютно (абс. тут). """ DEFAULT_EXCLUDE_DIRS = {

Цитирования: **0,01%** id: **125**

".git",

Цитирования: **0,01%** id: **126**

".hg",

Цитирования: **0,01%** id: **127**

".svn",

Цитирования: **0,01%** id: **128**

"__pycache__",

Цитирования: **0,01%** id: **129**

".mypy_cache",

Цитирования: **0,01%** id: **130**

".pytest_cache",

Цитирования: **0,01%** id: **131**

".tox",

Цитирования: **0,01%** id: **132**

"venv",

Цитирования: **0,01%** id: **133**

".venv",

Цитирования: **0,01%** id: **134**

"build",

Цитирования: **0,01%** id: **135**

"dist"

} def scan(self, project_path: str, config: Optional[Dict[str, Any]] = None) - List[str]: root = Path(project_path).resolve() if not root.exists() or not root.is_dir(): raise FileNotFoundError(f

Цитирования: **0,06%** id: **136**

"Project path not found or not a directory: {root}"

) cfg = config or {} exclude_dirs = set(cfg.get(

Цитирования: **0,01%** id: **137**

"exclude_dirs", []

)) | set(self.DEFAULT_EXCLUDE_DIRS) exclude_globs = list(cfg.get(

Цитирования: **0,01%** id: **138**

"exclude_globs", []

)) # e.g. [

Цитирования: **0,01%** id: **139**

"*/migrations/*",

Цитирования: **0,01%** id: **140**

"*/generated/*"

] include_globs = list(cfg.get("include_globs", ["**/*.py"])) files: List[str] = [] for pattern in include_globs: for p in root.glob(pattern): if not p.is_file(): continue # Exclude by directory name parts = set(p.parts) if any(d in parts for d in exclude_dirs): continue # Exclude by glob patterns (fnmatch over POSIX string) posix = p.as_posix() if any(fnmatch.fnmatch(posix, g) for g in exclude_globs): continue files.append(str(p)) files.sort() return files # ========== # ASTAnalyzer # ========== class ASTAnalyzer: "

Цитирования: **0%** id: **141**

""

Парсить *.py файли в AST і витягує залежності імпортів у FactsStore. """ def analyze(self, files: List[str], project_root: str) - FactsStore: store = FactsStore(project_root=str(Path(project_root).resolve())) root = Path(store.project_root) for file_path in files: path = Path(file_path) try: code = path.read_text(encoding=

Цитирования: **0,01%** id: **142**

"utf-8"

) except UnicodeDecodeError: # fallback: try latin-1 code = path.read_text(encoding=

Цитирования: **0,01%** id: **143**

"latin-1"

) try: tree = ast.parse(code, filename=str(path)) except SyntaxError as e: store.add_parse_error(str(path), f

Цитирования: **0,04%** id: **144**

"SyntaxError: {e.msg} (line {e.lineno})"

) continue src_module = _module_name_from_path(root, path) visitor = _ImportVisitor(src_module=src_module, src_file=str(path), store=store, project_root=root) visitor.visit(tree) return store class _ImportVisitor(ast.NodeVisitor): def __init__(self, src_module: str, src_file: str, store: FactsStore, project_root: Path) - None: self.src_module = src_module self.src_file = src_file self.store = store self.project_root = project_root def visit_Import(self, node: ast.Import) - Any: for alias in node.names: dst = alias.name # e.g.

Цитирования: **0,01%** id: **145**

"package.module"

self.store.add_import(ImportFact(src_module=self.src_module, dst_module=dst, src_file=self.src_file, lineno=getattr(node,

Цитирования: **0,01%** id: **146**

"lineno",

0) or 0, col_offset=getattr(node,

Цитирования: **0,01%** id: **147**

"col_offset",

0) or 0, kind=

Цитирования: **0,01%** id: **148**

"Import",

)) self.generic_visit(node) def visit_ImportFrom(self, node: ast.ImportFrom) - Any: # from X import a, b OR from .sub import a base = node.module or

Цитирования: **0%** id: **149**

""

level = getattr(node,

Цитирования: **0,01%** id: **150**

"level",

0) or 0 dst = _resolve_from_import(self.src_module, base, level) # If dst empty (e.g.

Цитирования: **0,02%** id: **151**

"from . import x"

where module is None) if dst: self.store.add_import(ImportFact(src_module=self.src_module, dst_module=dst, src_file=self.src_file, lineno=getattr(node,

Цитирования: **0,01%** id: **152**

"lineno",

0) or 0, col_offset=getattr(node,

Цитирования: **0,01%** id: **153**

"col_offset",

0) or 0, kind=

Цитирования: **0,01%** id: **154**

"from",

)) self.generic_visit(node) def _module_name_from_path(project_root: Path, file_path: Path) - str: """ Перетворює шлях .../pkg/sub/mod.py у 'pkg.sub.mod' Для .../pkg/sub/__init__.py - 'pkg.sub' Якщо не в межах project_root, повертає stem. """ try: rel = file_path.resolve().relative_to(project_root.resolve()) except Exception: return file_path.stem parts = list(rel.parts) if not parts: return file_path.stem # remove suffix .py if parts[-1].endswith(

Цитирования: **0,03%** id: **155**

".py"): parts[-1] = parts[-1]

[:-3] # __init__ - remove last part if parts[-1] ==

Цитирования: **0,03%** id: **156**

"__init__": parts = parts[:-1]

filter empty parts = [p for p in parts if p] return

Цитирования: **0%** id: **157**

".".

join(parts) if parts else file_path.stem def _resolve_from_import(src_module: str, base: str, level: int) - str: """ Розв'язує 'from ... import ...' у назву модулю для залежності. Для рівня level 0 виконує підйом по src_module. Приклади: src_module='app.api.routes', base='app.domain', level=0 - 'app.domain' src_module='app.api.routes', base='services', level=1 - 'app.api.services' src_module='app.api.routes', base='', level=1 (from . import x) - 'app.api' """ if level = 0: return

base src_parts = src_module.split(

Цитирования: **0,03%** id: **158**

".") if src_module else []

level=1 означає

Цитирования: **0,03%** id: **159**

"в межах поточного пакета":

піднятись на 1 сегмент (модуль) - пакет # рівень підйому: level up = min(level, len(src_parts)) prefix_parts = src_parts[:-up] if up 0 else src_parts if base: return

Цитирования: **0%** id: **160**

".".

join(prefix_parts + base.split(

Цитирования: **0%** id: **161**

"."

)) return

Цитирования: **0%** id: **162**

".".

join(prefix_parts) # ========== # GraphBuilder # ========== @dataclass class DependencyGraph: """ Мінімальний орієнтований граф залежностей. nodes: множина модулів/пакетів edges: u - v edge_sources: мапа (u,v) - список місць у коді (файл,рядок) """ nodes: Set[str] = field(default_factory=set) edges: Set[Tuple[str, str]] = field(default_factory=set) edge_sources: Dict[Tuple[str, str], List[Tuple[str, int]]] = field(default_factory=lambda: defaultdict(list)) def add_edge(self, u: str, v: str, src_file: str =

Цитирования: **0%** id: **163**

"",

lineno: int = 0) - None: self.nodes.add(u) self.nodes.add(v) self.edges.add((u, v)) if src_file: self.edge_sources[(u, v)].append((src_file, lineno)) def out_neighbors(self, u: str) - List[str]: return [v for (x, v) in self.edges if x == u] def in_neighbors(self, v: str) - List[str]: return [u for (u, y) in self.edges if y == v] class GraphBuilder: """ Будує граф залежностей з фактів (імпорти). Підтримує: - internal_only: відкидати зовнішні залежності (не з префіксами internal_roots) - level: 'module' або 'package' (згортання) """ def build(self, facts: FactsStore, config: Optional[Dict[str, Any]] = None) - DependencyGraph: cfg = config or {} internal_roots: List[str] = cfg.get(

Цитирования: **0,01%** id: **164**

"internal_roots", []

) # e.g. [

Цитирования: **0,01%** id: **165**

"app",

Цитирования: **0,01%** id: **166**

"src",

Цитирования: **0,01%** id: **167**

"myproj"

] internal_only: bool = bool(cfg.get(

Цитирования: **0,01%** id: **168**

"internal_only",

True)) level: str = cfg.get(

Цитирования: **0,01%** id: **169**

"graph_level",

Цитирования: **0,01%** id: **170**

"module"

) #

Цитирования: **0,01%** id: **171**

"module"

|

Цитирования: **0,01%** id: **172**

"package"

g = DependencyGraph() for imp in facts.imports: u = imp.src_module v = imp.dst_module if level ==

Цитирования: **0,01%** id: **173**

"package":

u = _to_package(u) v = _to_package(v) if internal_only and internal_roots: if not _is_internal(u, internal_roots) or not _is_internal(v, internal_roots): continue # remove self-edge from package collapse noise (optional) if u == v: continue g.add_edge(u, v, src_file=imp.src_file, lineno=imp.lineno) return g def _to_package(module_name: str) - str:

Цитирования: **0,07%** id: **174**

"""'a.b.c' - 'a.b' (умовно), 'a' - 'a'."""

parts = module_name.split(

Цитирования: **0%** id: **175**

"."

) return

Цитирования: **0,01%** id: **176**

".".join(parts[:-1]

) if len(parts) 1 else module_name def _is_internal(module_name: str, roots: Sequence[str]) - bool: return any(module_name == r or module_name.startswith(r +

Цитирования: **0%** id: **177**

"."

) for r in roots) # ========== # RuleEngine # ========== @dataclass(frozen=True) class Violation: rule_id: str rule_type: str message: str src: str dst: str evidence: List[Tuple[str, int]] = field(default_factory=list) class RuleEngine: """ Підтримувані правила (узагальнений DSL через dict): - forbid_edges: заборона залежностей між множинами - layers: шарова модель (forbid upward dependencies) - no_cycles: заборона циклів (SCC) - forbid_reachability: транзитивна заборона шляхів Очікуваний формат rules (приклад): rules = {

Цитирования: **0,01%** id: **178**

"forbid_edges":

[{

Цитирования: **0,02%** id: **179**

"id":"R1",

Цитирования: **0,03%** id: **180**

"from":"app.domain.*",

Цитирования: **0,03%** id: **181**

"to":"Infrastructure.*",

Цитирования: **0,05%** id: **182**

"msg":"Domain must not depend on Infrastructure"

}],

Цитирования: **0,01%** id: **183**

"layers":

{

Цитирования: **0,02%** id: **184**

"id":"R2",

Цитирования: **0,01%** id: **185**

"order":

[

Цитирования: **0,01%** id: **186**

"api",

Цитирования: **0,01%** id: **187**

"app.application",

Цитирования: **0,01%** id: **188**

"app.domain",

Цитирования: **0,01%** id: **189**

"infrastructure"

],

Цитирования: **0,01%** id: **190**

"mode":

Цитирования: **0,01%** id: **191**

"down"

allow only edges from earlier - later? (або навпаки) },

Цитирования: **0,01%** id: **192**

"no_cycles":

{

Цитирования: **0,02%** id: **193**

"id":"R3",

Цитирования: **0,01%** id: **194**

"enabled":

True},

Цитирования: **0,01%** id: **195**

"forbid_reachability":

[{

Цитирования: **0,02%** id: **196**

"id":"R4",

Цитирования: **0,03%** id: **197**

"from":"api.*",

Цитирования: **0,03%** id: **198**

"to":"infrastructure.*",

Цитирования: **0,05%** id: **199**

"msg":"API must not reach infrastructure transitively"

}] } """ def check(self, graph: DependencyGraph, rules: Dict[str, Any], facts: Optional[Any] = None) - List[Dict[str, Any]]: violations: List[Violation] = [] # 1) Forbidden direct edges for r in rules.get(

Цитирования: **0,03%** id: **200**

"forbid_edges", []) or []

: violations.extend(self._check_forbid_edges(graph, r)) # 2) Layers layers_rule = rules.get(

Цитирования: **0,01%** id: **201**

"layers"

) if layers_rule: violations.extend(self._check_layers(graph, layers_rule)) # 3) No cycles no_cycles = rules.get(

Цитирования: **0,01%** id: **202**

"no_cycles",

{

Цитирования: **0,01%** id: **203**

"enabled":

False}) if isinstance(no_cycles, dict) and no_cycles.get(

Цитирования: **0,01%** id: **204**

"enabled"

): violations.extend(self._check_cycles(graph, no_cycles)) # 4) Forbidden reachability (transitive) for r in rules.get(

Цитирования: **0,03%** id: **205**

"forbid_reachability", []) or []

: violations.extend(self._check_forbid_reachability(graph, r)) return [v.__dict__ for v in violations] def _check_forbid_edges(self, graph: DependencyGraph, rule: Dict[str, Any]) - List[Violation]: rid = rule.get(

Цитирования: **0,01%** id: **206**

"id",

Цитирования: **0,01%** id: **207**

"FORBID_EDGES"

) frm = rule.get(

Цитирования: **0,01%** id: **208**

"from",

Цитирования: **0,01%** id: **209**

"*"

) to = rule.get(

Цитирования: **0,01%** id: **210**

"to",

Цитирования: **0,01%** id: **211**

"*"

) msg = rule.get(

Цитирования: **0,01%** id: **212**

"msg",

Цитирования: **0,03%** id: **213**

"Forbidden dependency") out: List[Violation]

= [] for (u, v) in graph.edges: if _match(u, frm) and _match(v, to): out.append(Violation(rule_id=rid, rule_type=

Цитирования: **0,01%** id: **214**

"forbid_edges",

message=f

Цитирования: **0,03%** id: **215**

"{msg}: {u} - {v}",

src=u, dst=v, evidence=list(graph.edge_sources.get((u, v), [])),)) return out def

_check_layers(self, graph: DependencyGraph, rule: Dict[str, Any]) -> List[Violation]: rid = rule.get(

Цитирования: **0,01%** id: **216**

"id",

Цитирования: **0,02%** id: **217**

"LAYERS") order: List[str]

= rule.get(

Цитирования: **0,01%** id: **218**

"order", []

) mode = rule.get(

Цитирования: **0,01%** id: **219**

"mode",

Цитирования: **0,01%** id: **220**

"down"

) #

Цитирования: **0,01%** id: **221**

"down":

allow i- j where i = j ;

Цитирования: **0,01%** id: **222**

"up":

allow i- j where i = j msg = rule.get(

Цитирования: **0,01%** id: **223**

"msg",

Цитирования: **0,01%** id: **224**

"Layering violation"

) # Build index: pattern - idx ; pattern matching is flexible: each order entry can be exact or wildcard layer_patterns = order idx_map = {pat: i for i, pat in enumerate(layer_patterns)} def layer_index(node: str) - Optional[int]: # first match in order for pat, i in idx_map.items(): if _match(node, pat): return i return None out: List[Violation] = [] for (u, v) in graph.edges: iu = layer_index(u) iv = layer_index(v) if iu is None or iv is None: continue if mode ==

Цитирования: **0,01%** id: **225**

"down":

allowed = iu = iv else: allowed = iu = iv if not allowed: out.append(Violation(rule_id=rid, rule_type=

Цитирования: **0,01%** id: **226**

"layers",

message=f

Цитирования: **0,05%** id: **227**

"{msg}: {u} (layer {iu}) - {v} (layer {iv})",

src=u, dst=v, evidence=list(graph.edge_sources.get((u, v), [])),)) return out def _check_cycles(self, graph: DependencyGraph, rule: Dict[str, Any]) -> List[Violation]: rid = rule.get(

Цитирования: **0,01%** id: **228**

"id",

Цитирования: **0,01%** id: **229**

"NO_CYCLES"

) msg = rule.get(

Цитирования: **0,01%** id: **230**

"msg",

Цитирования: **0,02%** id: **231**

"Cyclic dependency detected"

) sccs = strongly_connected_components(graph.nodes, graph.edges) out: List[Violation] = [] for comp in sccs: if len(comp) 1: nodes_sorted = sorted(comp) # Emit one violation per SCC (узагальнено) out.append(Violation(rule_id=rid, rule_type=

Цитирования: **0,01%** id: **232**

"no_cycles",

message=f

Цитирования: **0,05%** id: **233**

"{msg}: SCC size={len(comp)} nodes={nodes_sorted}", src=nodes_sorted[0]

, dst=nodes_sorted[-1], evidence=[],)) return out def _check_forbid_reachability(self, graph: DependencyGraph, rule: Dict[str, Any]) - List[Violation]: rid = rule.get(

Цитирования: **0,01%** id: **234**

"id",

Цитирования: **0,01%** id: **235**

"FORBID_REACHABILITY"

) frm = rule.get(

Цитирования: **0,01%** id: **236**

"from",

Цитирования: **0,01%** id: **237**

"*"

) to = rule.get(

Цитирования: **0,01%** id: **238**

"to",

Цитирования: **0,01%** id: **239**

"*"

) msg = rule.get(

Цитирования: **0,01%** id: **240**

"msg",

Цитирования: **0,02%** id: **241**

"Forbidden transitive dependency"

) # Precompute adjacency adj = defaultdict(list) for (u, v) in graph.edges: adj[u].append(v) sources = [n for n in graph.nodes if _match(n, frm)] targets = {n for n in graph.nodes if _match(n, to)} out: List[Violation] = [] for s in sources: reached = bfs_reach(adj, s) bad = sorted(list(set(reached) & targets)) if bad: # Узагальнено: одне порушення на (s, any target) out.append(Violation(rule_id=rid, rule_type=

Цитирования: **0,01%** id: **242**

"forbid_reachability",

message=f

Цитирования: **0,04%** id: **243**

"{msg}: {s} reaches {bad}", src=s, dst=bad[0]

, evidence=[],)) return out def _match(name: str, pattern: str) - bool: """ Підтримує: - shell wildcards: app.domain.* , infrastructure.* - exact names: app.domain - '*' for all """ if not pattern or pattern ==

Цитирования: **0,01%** id: **244**

"*":

return True return fnmatch.fnmatch(name, pattern) def bfs_reach(adj: Dict[str, List[str]], start: str)

- Set[str]: q = deque([start]) seen = {start} while q: u = q.popleft() for v in adj.get(u, []): if v not in seen: seen.add(v) q.append(v) seen.discard(start) return seen def strongly_connected_components(nodes: Iterable[str], edges: Iterable[Tuple[str, str]]) - List[Set[str]]: """ Tarjan SCC (без зовнішніх бібліотек). """ adj = defaultdict(list) for u, v in edges: adj[u].append(v) index = 0 indices: Dict[str, int] = {} lowlink: Dict[str, int] = {} stack: List[str] = [] onstack: Set[str] = set() sccs: List[Set[str]] = [] def strongconnect(v: str) - None: nonlocal index indices[v] = index lowlink[v] = index index += 1 stack.append(v) onstack.add(v) for w in adj.get(v, []): if w not in indices: strongconnect(w) lowlink[v] = min(lowlink[v], lowlink[w]) elif w in onstack: lowlink[v] = min(lowlink[v], indices[w]) if lowlink[v] == indices[v]: comp: Set[str] = set() while True: w = stack.pop() onstack.remove(w) comp.add(w) if w == v: break sccs.append(comp) for v in nodes: if v not in indices: strongconnect(v) return sccs # ============================ # MetricsEngine # ============================ class MetricsEngine: """ Обчислює базові метрики: - |V|, |E|, density - in/out degrees (avg, max) - SCC count, max SCC size - violations_total + by_type - top offenders (by src/dst participation) """ def compute(self, graph: DependencyGraph, violations: List[Dict[str, Any]], config: Optional[Dict[str, Any]] = None) - Dict[str, Any]: V = len(graph.nodes) E = len(graph.edges) density = (E / (V * (V - 1))) if V 1 else 0.0 out_deg = Counter() in_deg = Counter() for u, v in graph.edges: out_deg[u] += 1 in_deg[v] += 1 avg_out = (sum(out_deg.values()) / V) if V else 0.0 avg_in = (sum(in_deg.values()) / V) if V else 0.0 max_out = max(out_deg.values()) if out_deg else 0 max_in = max(in_deg.values()) if in_deg else 0 sccs = strongly_connected_components(graph.nodes, graph.edges) cyclic_sccs = [c for c in sccs if len(c) 1] scc_count = len(cyclic_sccs) max_scc_size = max((len(c) for c in cyclic_sccs), default=0) by_type = Counter(v.get(

Цитирования: **0,01%** id: **245**

"rule_type",

Цитирования: **0,01%** id: **246**

"unknown"

) for v in violations) total_violations = len(violations) participation = Counter() for v in violations: participation[v.get(

Цитирования: **0,01%** id: **247**

"src",

Цитирования: **0,01%** id: **248**

"")]

+= 1 participation[v.get(

Цитирования: **0,01%** id: **249**

"dst",

Цитирования: **0%** id: **250**

""

)] += 1 top_offenders = [{

Цитирования: **0,01%** id: **251**

"component":

k,

Цитирования: **0,01%** id: **252**

"count":

c} for k, c in participation.most_common(5) if k] return {

Цитирования: **0,01%** id: **253**

"graph":

{

Цитирования: **0,01%** id: **254**

"nodes":

V,

Цитирования: **0,01%** id: **255**

"edges":

E,

Цитирования: **0,01%** id: **256**

"density":

round(density, 6),

Цитирования: **0,01%** id: **257**

"avg_out_degree":

round(avg_out, 3),

Цитирования: **0,01%** id: **258**

"avg_in_degree":

round(avg_in, 3),

Цитирования: **0,01%** id: **259**

"max_out_degree":

int(max_out),

Цитирования: **0,01%** id: **260**

"max_in_degree":

int(max_in), },

Цитирования: **0,01%** id: **261**

"cycles":

{

Цитирования: **0,01%** id: **262**

"scc_count":

int(scc_count),

Цитирования: **0,01%** id: **263**

"max_scc_size":

int(max_scc_size), },

Цитирования: **0,01%** id: **264**

"violations":

{

Цитирования: **0,01%** id: **265**

"total":

int(total_violations),

Цитирования: **0,01%** id: **266**

"by_type":

dict(by_type),

Цитирования: **0,01%** id: **267**

"top_offenders":

top_offenders, }, } # ========== # Reporter # ========== class Reporter: """ Генерує звіти: - text: читабельний консольний - json: структурований (можна зберегти у файл) """ def render(self, result: Any, format: str =

Цитирования: **0,01%** id: **268**

"text",

output_path: Optional[str] = None, config: Optional[Dict[str, Any]] = None,) - str: fmt = (format or

Цитирования: **0,01%** id: **269**

"text"

).lower().strip() if fmt not in {

Цитирования: **0,01%** id: **270**

"text",

Цитирования: **0,01%** id: **271**

"json"

}: raise ValueError(f

Цитирования: **0,03%** id: **272**

"Unsupported report format: {format}"

) payload = {

Цитирования: **0,01%** id: **273**

"metrics":

getattr(result,

Цитирования: **0,01%** id: **274**

"metrics",

None),

Цитирования: **0,01%** id: **275**

"violations":

getattr(result,

Цитирования: **0,01%** id: **276**

"violations",

None), } if fmt ==

Цитирования: **0,01%** id: **277**

"json":

text = json.dumps(payload, ensure_ascii=False, indent=2) else: text = self._render_text(payload) if output_path: Path(output_path).write_text(text, encoding=

Цитирования: **0,01%** id: **278**

"utf-8"

) return text def _render_text(self, payload: Dict[str, Any]) - str: metrics = payload.get(

Цитирования: **0,01%** id: **279**

"metrics"

) or {} violations = payload.get(

Цитирования: **0,03%** id: **280**

"violations") or [] lines: List[str]

= [] lines.append(

Цитирования: **0,03%** id: **281**

"=== Archcon Report ==="

) lines.append(

Цитирования: **0%** id: **282**

""

) # Metrics g = metrics.get(

Цитирования: **0,01%** id: **283**

"graph",

{}) c = metrics.get(

Цитирования: **0,01%** id: **284**

"cycles",

{}) v = metrics.get(

Цитирования: **0,01%** id: **285**

"violations",

{}) lines.append(

Цитирования: **0,01%** id: **286**

"[Graph]"

) lines.append(f

Цитирования: **0,05%** id: **287**

"- Nodes: {g.get('nodes', 0)}"

) lines.append(f

Цитирования: **0,05%** id: **288**

"- Edges: {g.get('edges', 0)}"

) lines.append(f

Цитирования: **0,05%** id: **289**

"- Density: {g.get('density', 0)}"

) lines.append(f

Цитирования: **0,05%** id: **290**

"- Avg out-degree: {g.get('avg_out_degree', 0)}"

) lines.append(f

Цитирования: **0,05%** id: **291**

"- Avg in-degree: {g.get('avg_in_degree', 0)}"

) lines.append(f

Цитирования: **0,05%** id: **292**

"- Max out-degree: {g.get('max_out_degree', 0)}"

) lines.append(f

Цитирования: **0,05%** id: **293**

"- Max in-degree: {g.get('max_in_degree', 0)}"

) lines.append(

Цитирования: **0,02%** id: **294**

"") lines.append("[Cycles]"

) lines.append(f

Цитирования: **0,05%** id: **295**

"- SCC count: {c.get('scc_count', 0)}"

) lines.append(f

Цитирования: **0,06%** id: **296**

"- Max SCC size: {c.get('max_scc_size', 0)}"

) lines.append(

Цитирования: **0%** id: **297**

""

) lines.append(

Цитирования: **0,01%** id: **298**

"[Violations]"

) lines.append(f

Цитирования: **0,05%** id: **299**

"- Total: {v.get('total', 0)}"

) by_type = v.get(

Цитирования: **0,01%** id: **300**

"by_type",

{}) if by_type: lines.append("- By type:") for k, cnt in sorted(by_type.items(), key=lambda x: (-x[1], x[0])): lines.append(f" * {k}: {cnt}

Цитирования: **0,07%** id: **301**

") top = v.get("top_offenders", []) if top: lines.append("-

Top offenders:

Цитирования: **0,06%** id: **302**

") for item in top: lines.append(f" * {item['component']

}: {item['count']}

Цитирования: **0,04%** id: **303**

") lines.append("") if violations: lines.append("

Details:

Цитирования: **0,07%** id: **304**

") for i, viol in enumerate(violations, 1): lines.append(f"

{i}. [{viol.get('rule_id')}/{viol.get('rule_type')}] {viol.get('message')}

Цитирования: **0,16%** id: **305**

") ev = viol.get("evidence") or [] if ev: # show only first 3 evidence entries for (f, ln) in ev[:3]: lines.append(f"

- at {f}:{ln}

Цитирования: **0,02%** id: **306**

") else: lines.append("

No violations found.

Цитирования: **0,01%** id: **307**

") return "

\n

Цитирования: **0,32%** id: **308**

".join(lines) # ============================ # (Optional) Simple RulesConfig # ============================ # Якщо потрібно: YAML підтримка вимагатиме PyYAML. # Тут — мінімальний JSON-only loader як fallback. class JsonRulesConfig: def load(self, path: str) - Dict[str, Any]: p = Path(path) if not p.exists(): raise FileNotFoundError(f"

Config not found: {path}

Цитирования: **0,03%** id: **309**

") if p.suffix.lower() == ".

json

Цитирования: **0,05%** id: **310**

": return json.loads(p.read_text(encoding="

utf-8

Цитирования: **0,11%** id: **311**

")) # Якщо не JSON — повертаємо порожній (або кинути помилку). Для YAML додайте PyYAML. raise ValueError("

Only .json config is supported by JsonRulesConfig. Install PyYAML for YAML support.")

Заявление об ограничении ответственности:

Этот отчет должен быть правильно истолкован и проанализирован квалифицированным специалистом, который несет ответственность за оценку!

Любая информация, представленная в этом отчете, не является окончательной и подлежит ручному просмотру и анализу.
Пожалуйста, следуйте инструкциям: Рекомендации по оценке

Детектор Плагиата - Ваше право на оригинальность! SkyLine LLC