МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 61.00.00.000 ПЗ

Група ІПм-24-3

Гуцул Олександр

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Гуцул Олександр Тарасович
(прізвище, ім’я, по батькові)

УДК 004.9
(індекс)

МАГІСТЕРСЬКА РОБОТА

Формальні методи та інструментальні засоби забезпечення архітектурної цілісності програмних системах
(назва роботи)

Інженерія програмного забезпечення
(назва освітньої програми)

121 - Інженерія програмного забезпечення
(шифр і назва спеціальності)

Гуцул О.Т.
(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Бандура Вікторія Валеріївна, к.т.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту
Завідувач кафедри
доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль
доц. Вовк Р.Б.
(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2025 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Гуцулу Олександру Тарасовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Формальні методи та інструментальні засоби забезпечення архітектурної цілісності програмних системах ”

керівник проекту (роботи) Бандура В.В., к.т.н., доцент

затверджені наказом закладу вищої освіти від “ 05 ” листопада 2025 р. № 695/7

2. Строк подання студентом проекту (роботи) 25 січня 2026 р.

3. Вихідні дані до проекту (роботи) Формальні моделі і методи побудови інформаційних та програмних технологій цілісності архітектури систем

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Дослідження предметної області використання засобів забезпечення архітектурної цілісності
2. Математична формалізація архітектурної відповідності
3. Формальні методи та засоби емпіричних проявів архітектурної ерозії в програмних системах
4. Імплементація методів та алгоритмів забезпечення архітектурної цілісності програмних систем

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. MVC архітектура (рис. 1.1)
2. Представлення варіації архітектурних патернів MV* (рис. 1.2)
3. Архітектура аркадного та First Person Shooter проектів (рис. 1.3)
4. Архітектура Shoot 'Em Up проекту (рис. 1.4)
5. Архітектура Isometric RPG та 3D Platformer проектів (рис. 1.5)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2025 р.

Керівник ______________________________
(підпис)

Завдання прийняв до виконання______________________________
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2025	виконано
2	Дослідження предметної області використання засобів забезпечення архітектурної цілісності	01.10.2025	виконано
3	Математична формалізація архітектурної відповідності	22.10.2025	виконано
4	Формальні методи та засоби емпіричних проявів архітектурної ерозії в програмних системах	15.11.2025	виконано
5	Імплементація методів та алгоритмів забезпечення архітектурної цілісності програмних систем	03.12.2025	виконано
6	Реалізація методології оцінки ефективності методу	27.12.2025	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	25.01.2026	виконано

Студент – магістр ____________________
(підпис)

Керівник роботи ____________________
(підпис)

АНОТАЦІЯ

Магістерська робота: 77 с., 16 рис., 10 табл., 43 джерела.

Тема: Формальні методи та інструментальні засоби забезпечення архітектурної цілісності програмних системах

Мета роботи: розробка та обґрунтування формалізованого методу забезпечення архітектурної цілісності програмних систем на основі архітектурних шаблонів та статичних методів верифікації.

Об'єктом дослідження є процеси проєктування, еволюції та супроводу архітектури програмних систем.

Предметом дослідження є формальні методи, математичні моделі та інструментальні засоби верифікації архітектурної відповідності програмних систем.

Результати дослідження

В роботі запропоновано формалізований підхід до забезпечення архітектурної цілісності програмних систем на основі поєднання архітектурних шаблонів і методів статичної верифікації.

Висновок

Вдосконалено математичну модель перевірки архітектурної відповідності з використанням графової декомпозиції залежностей і розроблено класифікацію типових форм архітектурної ерозії для шаблону Model–View–Controller.

АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, АРХІТЕКТУРНА ЦІЛІСНІСТЬ, АРХІТЕКТУРНА ЕРОЗІЯ, ФОРМАЛЬНІ МЕТОДИ, СТАТИЧНИЙ АНАЛІЗ, АРХІТЕКТУРНІ ШАБЛОНИ, MODEL–VIEW–CONTROLLER, ВЕРИФІКАЦІЯ АРХІТЕКТУРИ, ТЕХНІЧНИЙ БОРГ

ABSTRACT

Master Thesis: 77 pp., 16 fig., 10 tab., 43 sources.

Topic: Formal methods and tools for ensuring architectural integrity in software systems

Purpose of the work: development and justification of a formalized method for ensuring architectural integrity of software systems based on architectural templates and static verification methods.

The object of the research is the processes of design, evolution and maintenance of the architecture of software systems.

The subject of the research is formal methods, mathematical models and tools for verifying the architectural compliance of software systems.

Research results

The paper proposes a formalized approach to ensuring architectural integrity of software systems based on a combination of architectural templates and static verification methods.

Conclusion

A mathematical model for verifying architectural compliance using graph decomposition of dependencies has been improved and a classification of typical forms of architectural erosion for the Model–View–Controller template has been developed.

SOFTWARE ARCHITECTURE, ARCHITECTURAL INTEGRITY, ARCHITECTURAL EROSION, FORMAL METHODS, STATIC ANALYSIS, ARCHITECTURAL TEMPLATES, MODEL–VIEW–CONTROLLER, ARCHITECTURAL VERIFICATION, TECHNICAL DEBT

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 9
ВСТУП 10

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ВИКОРИСТАННЯ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ЗАБЕЗПЕЧЕННЯ АРХІТЕКТУРНОЇ ЦІЛІСНОСТІ СИСТЕМИ 14
1.1. Наукове обґрунтування методу верифікації архітектурних шаблонів 14
1.2. Особливості проблеми архітектурної еволюції та цілісності 15
1.3. Концептуальні засади архітектури програмного забезпечення 17
1.3.1 Архітектурні шаблони як когнітивні одиниці 18
1.3.2 Методологія перевірки архітектурної відповідності (ACC) 19
1.3.3. Гібридний підхід та архітектурна верифікація 20
1.4. Математична формалізація архітектурної відповідності 21
1.4.1. Матричний підхід до аналізу 22
1.4.2. Класифікація виявлених аномалій 23
Висновки до розділу 23

РОЗДІЛ 2. ФОРМАЛЬНІ МЕТОДИ ТА ЗАСОБИ ЕМПІРИЧНИХ ПРОЯВІВ АРХІТЕКТУРНОЇ ЕРОЗІЇ В ПРОГРАМНИХ СИСТЕМАХ 25
2.1. Контекстуалізація шаблону Model-View-Controller 26
2.2. Ретроспективний аналіз архітектурної еволюції та ерозії шаблону MVC у розробці ігрових систем 28
2.3. Процес семантичного анотування (тегування) 33
2.4. Побудова графіку еволюції технічного боргу за жанрами проектів 35
2.5. Методологія статичної перевірки відповідності архітектури 37
2.5.1. Концептуальний базис SACC 39
2.5.2 Роль інспекцій у верифікації архітектури 40
2.5.3 Формалізація правил залежностей 41

2.6. Формулювання вимог та вибір методології SACC 41
Висновки до розділу 44

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ТА АЛГОРИТМІВ ЗАБЕЗПЕЧЕННЯ АРХІТЕКТУРНОЇ ЦІЛІСНОСТІ ПРОГРАМНИХ СИСТЕМ 46
3.1. Формалізація методу та аналіз порушень 46
3.1.1. Алгоритмічна база методу 46
3.1.2. Класифікація типових архітектурних ерозій у MVC 49
3.2. Математична модель перевірки залежностей 49
3.3. Деталізація методу формалізованої верифікації архітектурних інваріантів програмних систем на основі графової декомпозиції залежностей 51
3.4. Порівняльний аналіз технік моделювання рефлексії та правил залежностей 53
3.4.1 Метод SACC на основі архітектурних шаблонів 54
3.4.2 Верифікація через гіпотетичні сценарії 56
3.5. Методологія оцінки ефективності методу 61
3.6. Експериментальна оцінка методу 63
3.6.1 Процедура класифікації та верифікації 65
3.6.2 Рефакторинг та усунення ерозії 66
3.6.3 Вплив на структурні метрики 67
Висновки до розділу 69

ВИСНОВКИ 70
ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 73

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ArchC — Architectural Classification

DevC — Developer Classification

DivAbs — Divergence and Absence detection function

DSM — Dependency Structure Matrix

JITACC — Just-In-Time Architecture Conformance Checking

LINQ — Language Integrated Query

LoC — Lines of Code

MVC — Model-View-Controller

SACC — Static Architecture Conformance Checking

TP — True Positive

UIAPI — User Interface Application Programming Interface

VCS — Version Control System

ВСТУП

Актуальність теми.

Сучасні програмні системи характеризуються зростаючою складністю, тривалим життєвим циклом та постійною еволюцією архітектурних рішень. У процесі розвитку програмного забезпечення відбувається поступове відхилення фактичної реалізації від початкової архітектурної моделі, що призводить до виникнення явища архітектурної ерозії. Наслідками цього процесу є зниження зрозумілості системи, ускладнення супроводу, накопичення технічного боргу та зменшення якості програмного продукту в цілому.

Традиційні підходи до контролю архітектури, які ґрунтуються на ручних інспекціях або неформалізованих рекомендаціях, виявляються недостатніми для великих і динамічних програмних систем. У зв’язку з цим зростає потреба у використанні формальних методів та інструментальних засобів, здатних забезпечити систематичну перевірку архітектурної відповідності на всіх етапах життєвого циклу програмного забезпечення. Особливого значення набуває застосування архітектурних шаблонів як формалізованих моделей, що задають інваріанти структури та взаємодії компонентів системи.

У роботі розглядається проблема забезпечення архітектурної цілісності програмних систем на основі формалізованої верифікації архітектурних інваріантів. Основну увагу приділено поєднанню математичних моделей, статичних методів аналізу та архітектурних шаблонів, зокрема Model–View–Controller, як базису для контролю архітектурної еволюції. Запропонований підхід орієнтований на виявлення та усунення порушень архітектурної відповідності, що дозволяє підвищити якість та керованість програмних систем.

Актуальність дослідження зумовлена необхідністю підвищення надійності та підтримуваності програмних систем в умовах швидких змін

вимог та технологій. Архітектурна цілісність є одним із ключових факторів, що визначає здатність програмного забезпечення до масштабування, повторного використання та довготривалої еволюції. Водночас у практиці розробки програмного забезпечення відсутні універсальні формалізовані засоби, які дозволяли б систематично контролювати відповідність реалізації задекларованій архітектурі.

Проблема архітектурної ерозії особливо загострюється у проєктах з тривалим життєвим циклом, багатокомандною розробкою та високою динамікою змін. У таких умовах порушення архітектурних інваріантів часто залишаються непоміченими на ранніх етапах, що призводить до значних витрат на супровід та рефакторинг. Використання формальних методів перевірки архітектури дозволяє своєчасно виявляти відхилення та запобігати накопиченню технічного боргу.

Застосування інструментальних засобів статичної архітектурної верифікації, зокрема методів SACC, відкриває можливості для інтеграції архітектурного контролю в процеси безперервної розробки та тестування. Таким чином, дослідження, спрямоване на формалізацію та практичну реалізацію методів забезпечення архітектурної цілісності, є актуальним як з наукової, так і з прикладної точок зору.

Метою магістерської роботи є розробка та обґрунтування формалізованого методу забезпечення архітектурної цілісності програмних систем на основі архітектурних шаблонів та статичних методів верифікації.

Об’єктом дослідження є процеси проєктування, еволюції та супроводу архітектури програмних систем.

Предметом дослідження є формальні методи, математичні моделі та інструментальні засоби верифікації архітектурної відповідності програмних систем.

Завдання дослідження

Для досягнення поставленої мети у роботі необхідно вирішити такі завдання:

1. Проаналізувати сучасні підходи до забезпечення архітектурної цілісності програмних систем.

2. Дослідити причини та форми прояву архітектурної ерозії в процесі еволюції програмного забезпечення.

3. Формалізувати архітектурні шаблони як систему інваріантів і правил залежностей.

4. Запропонувати алгоритмічний метод виявлення порушень архітектурних інваріантів.

5. Провести експериментальну оцінку ефективності запропонованого методу.

Методи дослідження

У роботі використано такі методи дослідження: методи формального аналізу та математичного моделювання; теорію графів і матричні методи аналізу залежностей; методи статичного аналізу програмного коду; архітектурні інспекції та семантичне анотування компонентів; експериментальні методи оцінки ефективності програмних засобів; порівняльний аналіз архітектурних підходів і шаблонів.

Наукова новизна роботи полягає в тому, що:

- запропоновано формалізований підхід до забезпечення архітектурної цілісності програмних систем на основі поєднання архітектурних шаблонів і методів статичної верифікації;

- удосконалено математичну модель перевірки архітектурної відповідності з використанням графової декомпозиції залежностей;

- розроблено класифікацію типових форм архітектурної ерозії для шаблону Model–View–Controller;

- отримало подальший розвиток застосування методології SACC у контексті контролю архітектурної еволюції.

Практичне застосування результатів

Практичне значення отриманих результатів полягає у можливості використання запропонованого методу для автоматизованого контролю

архітектурної відповідності програмних систем. Результати роботи можуть бути застосовані під час проєктування, супроводу та рефакторингу програмного забезпечення, а також інтегровані в процеси безперервної інтеграції та аналізу якості коду. Запропонований підхід може бути використаний у навчальному процесі під час викладання дисциплін з архітектури програмного забезпечення та формальних методів.

Структура магістерської роботи. Представлена робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 77 сторінок, і містить 16 рисунків, 10 таблиці, перелік використаних джерел із 43 найменувань.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ВИКОРИСТАННЯ ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ЗАБЕЗПЕЧЕННЯ АРХІТЕКТУРНОЇ ЦІЛІСНОСТІ СИСТЕМИ

1.1. Наукове обґрунтування методу верифікації архітектурних шаблонів

Патерни проектування (шаблони) виступають фундаментальними когнітивними одиницями та джерелом експертних знань у процесі розробки складних програмних систем. Вони репрезентують високорівневі абстракції та валідовані практикою архітектурні рішення, що детермінують структурну організацію системи для забезпечення заданих атрибутів якості (гнучкості, масштабованості, підтримуваності).

Однак у процесі ітеративної розробки виникає явище архітектурної ерозії (software entropy). Незначні дефекти на рівні вихідного коду, накопичуючись, спричиняють деградацію структурних зв’язків, що призводить до невідповідності фактичної реалізації проектному замислу. Існуючі методи статичного аналізу переважно фокусуються на низькорівневих залежностях, не забезпечуючи необхідного рівня абстракції для цілісної верифікації шаблону.

У даній роботі представлено метод формалізованої верифікації, що дозволяє підвищити рівень абстракції від аналізу атомарних зв’язків до перевірки інваріантів ключових залежностей у межах обраного шаблону.

Практична реалізація методу була апробована на архітектурному шаблоні Model-View-Controller (MVC). Процес дослідження включав:

- Діагностику: виявлення архітектурних невідповідностей у реальних програмних системах.

- Рефакторинг: оцінку впливу усунення виявлених ерозій на якісні показники коду.

- Експериментальну валідацію: порівняльний аналіз контрольних груп.

Для оцінки ефективності запропонованого підходу було проведено контрольований експеримент у виробничому середовищі. Випадковий розподіл команд на експериментальну (використання автоматизованого сервісу моніторингу) та контрольну групи дозволив отримати статистично значущі дані.

Таблиця 1.1.

Параметри порівняння пропонованого підходу

Параметр порівняння	**Контрольна група (традиційна розробка)**	**Експериментальна група (з впровадженням методу)**
Частота архітектурних порушень	Висока (накопичувальна)	Низька (стабілізована)
Відповідність MVC-структурі	Деградує з часом	Підтримується на високому рівні
Час на виявлення ерозії	Відкладено (етап ревізії коду)	У реальному часі (Continuous Inspection)

Результати підтверджують гіпотезу: розробники, інтегровані в екосистему автоматизованої перевірки архітектурних обмежень, демонструють суттєво нижчий рівень деградації структури проекту. Це свідчить про доцільність впровадження засобів «архітектурного нагляду» в цикли безперервної інтеграції (CI/CD).

1.2. Особливості проблеми архітектурної еволюції та цілісності

Програмне забезпечення (ПЗ) за своєю природою є нематеріальним, адаптивним та динамічним артефактом. Саме властивість гнучкості (flexibility) виступає фундаментальною диференційною ознакою ПЗ порівняно з іншими антропогенними системами. Проте, попри іманентну пластичність коду, процес ітеративної модифікації та еволюції складних систем часто супроводжується значними труднощами. Акумуляція дефектів

проектування, невідповідностей та випадкових девіацій у вихідному коді призводить до стрімкого зростання часових та ресурсних витрат на впровадження змін. За таких умов ПЗ втрачає здатність до ефективного розвитку.

Здатність системи до еволюції детермінується її архітектурними характеристиками. Неконтрольоване накопичення технічного боргу спричиняє деградацію цільової структури — явище, відоме як архітектурна ерозія.

Для нівелювання процесів ерозії протягом усього життєвого циклу програмних систем необхідні релевантні методи та інструменти верифікації. Метою даного дослідження є розробка методу статичної перевірки відповідності архітектури (Static Architecture Conformance Checking, SACC). Попри існування аналогічних підходів, їхнє широке впровадження в індустрії обмежене високим порогом входження та складністю інструментарію. Наша концепція базується на використанні архітектурних шаблонів як фундаментальних одиниць перевірки, що дозволяє підвищити рівень абстракції від низькорівневих правил залежностей до високорівневих архітектурних конструкцій. Пріоритетом методу є когнітивна простота та високий рівень абстракції, що є критично важливим для промислового застосування.

Протягом життєвого циклу програмних систем вимоги та операційне середовище зазнають непередбачуваних трансформацій [13]. Репрезентативним прикладом останнього десятиліття є масова міграція сервісів на мобільні платформи, що висуває радикально нові вимоги до архітектури з точки зору технічних обмежень та взаємодії з користувачем. Дисципліна еволюції ПЗ вивчає закономірності адаптації систем до динамічних змін реального світу [11, 20]. Оскільки передбачити всі вектори розвитку на етапі ініціації проекту неможливо, архітектурна гнучкість стає стратегічним ресурсом.

В межах методологій гнучкої розробки (Agile) здатність до безперервної еволюції є ключовим фактором конкурентоспроможності [12]. У сучасному середовищі, де ПЗ є повсюдним (ubiquitous computing), роль еволюційної придатності лише зростає [11]. Платформи Web-орієнтованих систем, ігрові консолі та вбудовані автомобільні системи вимагають високої архітектурної пластичності, оскільки оновлення ПЗ (over-the-air updates) стали основним механізмом збільшення вартості та життєвого циклу продукту [5].

Фундаментальні закони еволюції ПЗ стверджують, що системи потребують постійної адаптації для підтримання рівня задоволеності користувачів [10]. Серед восьми законів [11] для даного дослідження найбільш релевантними є:

- Закон зростання складності: у процесі розвитку системи її складність зростає, якщо не вживаються спеціальні заходи з її мінімізації.

- Закон зниження якості: якість системи неминуче деградує, якщо її структура не адаптується до змін середовища.

Управління цими процесами через систематичну перевірку архітектурної відповідності є необхідною умовою підтримання життєздатності складних програмних проектів.

1.3. Концептуальні засади архітектури програмного забезпечення

Архітектура програмного забезпечення (АПЗ) є фундаментальною категорією програмної інженерії, спрямованою на декомпозицію та структурування складних систем для забезпечення їхньої керованості на високому рівні абстракції [1]. АПЗ виступає інтеграційним ядром, що визначає процеси збору вимог, проектування, верифікації та розгортання. Ключовим завданням архітектурного проектування є прийняття стратегічних рішень на ранніх етапах життєвого циклу системи на основі аналізу компромісів між суперечливими атрибутами якості.

У межах даного дослідження основна увага приділяється структурній архітектурі [2], яка визначає принципи поділу вихідного коду на логічні одиниці та регулює зв’язки між ними. Структурну архітектуру можна представити як сукупність кластерів програмних елементів (модулів), кожен з яких наділений специфічною зоною відповідальності (наприклад, рівень представлення даних). Формалізація цих кластерів — архітектурних елементів — дозволяє встановити регламент дозволених та заборонених залежностей, забезпечуючи такі характеристики системи, як модифікованість, повторне використання та поділ праці. Для об’єктно-орієнтованих систем структурна архітектура традиційно документується за допомогою уніфікованої мови моделювання (UML).

1.3.1 Архітектурні шаблони як когнітивні одиниці

Проектування АПЗ спирається на акумульований експертний досвід, систематизований у формі архітектурних шаблонів. Шаблон визначається як абстрактна пара «проблема–рішення», де проблема описує контекст і обмеження, а рішення пропонує інваріантну структуру для їх подолання.

Шаблони (наприклад, MVC та його похідні, такі як MVVM) слугують стратегічним інструментом мінімізації ризиків ерозії, оскільки вони пропонують валідовані часом механізми забезпечення еволюційної придатності ПЗ.

У процесі тривалої експлуатації та модифікації ПЗ спостерігається явище архітектурної ерозії — поступової втрати цілісності початкової структури внаслідок накопичення невідповідностей між проектною документацією та фактичною реалізацією. У науковій літературі цей процес описується через такі концепти, як архітектурний занепад [x30], програмна ентропія та технічний борг.

Ерозія призводить до девальвації архітектурних рішень: початкові компроміси втрачають актуальність, а складність супроводу та впровадження нових функцій зростає експоненціально. Основними стратегіями боротьби з

ерозією є запобігання, мінімізація та рефакторинг (видалення) виявлених дефектів.

1.3.2 Методологія перевірки архітектурної відповідності (ACC)

Процес верифікації відповідності фактичної реалізації системи її цільовій структурі визначається як Architecture Conformance Checking (ACC). Дана стратегія є проактивним методом мінімізації ерозії.

У роботі акцентується увага на статичній перевірці відповідності (SACC), яка передбачає аналіз артефактів вихідного коду без виконання програми. Незважаючи на інтенсивний розвиток методів SACC у науковому середовищі, рівень їх промислового впровадження залишається низьким [14]. Гіпотеза даного дослідження полягає в тому, що підвищення рівня абстракції верифікації до рівня архітектурних шаблонів сприятиме легшій інтеграції інструментарію в процеси безперервної інтеграції (CI/CD).

Дане дослідження базується на досвіді у сфері розробки складних програмних продуктів, зокрема ігрових систем. Специфіка ігрової індустрії вимагає екстремальної гнучкості архітектури для адаптації під суб'єктивні вимоги («ігровий досвід»), що робить цю галузь репрезентативним полігоном для вивчення еволюції ПЗ. Попередні спроби кількісно оцінити якість дизайну через метрики коду виявили обмеженість такого підходу через низьку стійкість метрик до міжпроектного порівняння. Це зумовило зміщення вектору дослідження у бік архітектурної абстракції та простоти. Замість деталізованого аналізу мікро-характеристик коду, пропонується метод візуалізації та контролю стратегічних архітектурних рішень, що робить вплив окремих розробників на цілісність системи прозорим та вимірюваним.

Для кращого розуміння методів контролю архітектурної відповідності важливо розмежувати підходи, що базуються на аналізі вихідного коду, та ті, що досліджують поведінку системи під час виконання.

Нижче наведено порівняльний аналіз статичної (SACC) та динамічної (DACC) перевірки відповідності архітектури.

Таблиця 1.1.

Порівняння SACC та DACC

Характеристика	Статичний аналіз (SACC)	Динамічний аналіз (DACC)
Об'єкт аналізу	Вихідний код, байт-код, конфігураційні файли.	Стан системи під час виконання (Runtime), трасування викликів.
Час проведення	Етап розробки, Code Review, CI/CD збірка.	Етап тестування, стейджинг або промислова експлуатація.
Рівень покриття	**Високий:** аналізуються всі можливі шляхи та залежності в коді.	**Залежний від тестів:** аналізуються лише ті гілки, що були активовані під час тестування.
Виявлення порушень	Виявляє структурні ерозії (заборонені імпорти, порушення шарів).	Виявляє непередбачені взаємодії між компонентами (наприклад, через рефлексію).
Ресурсні витрати	Низькі: не потребує розгортання та роботи всієї системи.	Високі: потребує налаштування середовища та сценаріїв навантаження.
Зворотний зв'язок	Негайний (до моменту злиття коду).	Відкладений (після запуску та проходження тестів).

1.3.3. Гібридний підхід та архітектурна верифікація

У дослідженнях SACC часто вважається пріоритетним для боротьби з ерозією на ранніх етапах, оскільки він дозволяє формалізувати архітектурні інваріанти як частину модульного тестування.

Особливості SACC:

- Фокус на структурі дозволяє перевірити відповідність діаграмі компонентів або пакетів.

- Математична точність. Як було розглянуто раніше, базується на порівнянні графів залежностей.

- Обмеження. Може не бачити залежностей, що виникають динамічно (через Dependency Injection або конфігурацію в БД).

Особливості DACC:

- Фокус на взаємодії. Ідеально підходить для верифікації мікросервісів та розподілених систем, де фактичні зв'язки встановлюються через мережу.

- Реальність. Показує фактичний потік даних, що може відрізнятися від того, що розробник "написав у коді".

- Обмеження. Складність автоматизації та ризик пропустити порушення в рідко використовуваних ділянках коду.

1.4. Математична формалізація архітектурної відповідності

Процес порівняння базується на аналізі гомоморфізму між графом реалізації та графом шаблону.

1. Формальне визначення графів

Нехай програмна система представлена як граф реалізації G_{impl} = (Vi ,Ei), де:

Vi — множина програмних модулів (класів, компонентів).

$Ei \subseteq Vi \times Vi$ — множина фактичних залежностей (виклик методу, доступ до даних тощо).

Нехай граф архітектурного шаблону (еталон) представлений як G_{spec}= (Vs,Es), де:

Vs — множина архітектурних ролей (наприклад, {Model, View, Controller}).

$Es \subseteq Vs \times Vs$ — множина дозволених взаємодій між ролями.

2. Функція мапінгу (Mapping Function)

Для перевірки відповідності необхідно визначити функцію відображення $f: Vi \rightarrow Vs$, яка призначає кожному модулю коду певну архітектурну роль.

- Якщо модуль $v \in Vi$ не має призначеної ролі, він вважається архітектурно нейтральним (або допоміжним).

- Якщо декілька модулів відображаються на одну роль, вони формують архітектурний шар.

3. Оператор виявлення ерозії

Архітектурна ерозія H визначається як множина фактичних залежностей, що не мають прообразу в специфікації. Формально це виражається через предикат:

$$H = \{(u, v) \in E_i \mid (f(u), f(v)) \notin E_s\}$$

Тобто, якщо в коді існує зв'язок між модулем u та v, але архітектурний шаблон забороняє взаємодію між їхніми ролями f(u) та f(v), то пара (u,v) класифікується як порушення (violation).

1.4.1. Матричний підхід до аналізу

Для автоматизації обчислень у великих системах зручно використовувати матриці суміжності.

1. Матриця реалізації (A_{impl}): Розміром n×n (де n — кількість класів), де $a_{jk}=1$, якщо клас j залежить від класу k.

2. Матриця специфікації (A_{spec}): Розміром m×m (де m — кількість ролей), що описує дозволені зв'язки.

3. Матриця мапінгу (T): Прямокутна матриця n×m, де $t_{jr}=1$, якщо клас j належить ролі r.

Формула виявлення порушень. Фактична архітектурна структура, отримана з коду, розраховується як:

$$A_{actual} = T^T \times A_{impl} \times T$$

Результат A_{actual} показує кількість або наявність зв'язків між ролями. Порівнюючи A_{actual} із A_{spec}, ми отримуємо матрицю ерозії:

$$Erosion = A_{actual} \circ (\neg A_{spec})$$

(де ∘ — оператор Адамара (поелементний добуток), а ¬ — логічне заперечення).

1.4.2. Класифікація виявлених аномалій

На основі цього апарату ми виділяємо два типи відхилень:

1. Divergence (Дивергенція).

Існує залежність у коді, яка заборонена в шаблоні (наприклад, Model → View). Математично: $(f(u),f(v)) \in A_{actual}$, але $(f(u),f(v)) \notin A_{spec}$.

2. Absence (Відсутність).

У шаблоні передбачено зв'язок, який не реалізований у коді. Хоча це не завжди є помилкою, це може свідчити про неповну реалізацію паттерна.

Такий підхід дозволяє підняти рівень аналізу від «рядків коду» до «алгебри архітектурних зв’язків», що робить метод SACC математично обґрунтованим та придатним для автоматизації.

Висновки до розділу

У першому розділі здійснено системний аналіз теоретичних засад забезпечення архітектурної цілісності програмних систем. Розглянуто еволюцію підходів до верифікації архітектурних шаблонів та обґрунтовано необхідність використання формальних методів у сучасних програмних проєктах. Показано, що проблема архітектурної ерозії є наслідком тривалої еволюції систем без належного контролю архітектурних інваріантів. Архітектурні шаблони інтерпретовано як когнітивні структури, що формалізують очікувану організацію програмних компонентів.

Проаналізовано методологію Architectural Compliance Checking як основу для виявлення невідповідностей між проєктною та реалізаційною архітектурою. Обґрунтовано доцільність гібридного підходу, який поєднує формальні, аналітичні та експертні методи перевірки. Запропоновано математичну модель архітектурної відповідності з використанням

матричного апарату. Здійснено класифікацію типових архітектурних аномалій та порушень залежностей.

Доведено, що формалізація архітектури підвищує керованість її еволюції. Отримані результати сформували теоретичне підґрунтя для подальших емпіричних і прикладних досліджень.

РОЗДІЛ 2. ФОРМАЛЬНІ МЕТОДИ ТА ЗАСОБИ ЕМПІРИЧНИХ ПРОЯВІВ АРХІТЕКТУРНОЇ ЕРОЗІЇ В ПРОГРАМНИХ СИСТЕМАХ

Першочерговим завданням даного етапу дослідження є верифікація гіпотези про те, що архітектурна ерозія є деструктивним чинником у реальних виробничих умовах. Особлива увага приділяється питанню, чи є імплементація архітектурного шаблону достатньою умовою для детермінованого контролю зростання складності в межах еволюційного циклу програмного забезпечення.

Об’єктом аналізу виступає досвід розробки ігрових систем, де архітектурне проектування відіграє критичну роль. Бло виявлено, що команди розробників систематично звертаються до шаблонів проектування для вирішення специфічного спектру інженерних викликів.

До них належать:

- керування надлишковою кодовою базою користувацьких інтерфейсів (UI),
- адаптація до динамічних змін у технологіях рендерингу реального часу,
- забезпечення гнучкості ігрових правил («ігрова механіка»),
- обробка великих масивів ігрових даних через спеціалізоване інструментарій (редактори рівнів).

Окрім технічних аспектів, архітектурна уніфікація спрямована на отримання організаційних преференцій: оптимізацію ротації розробників між проектами та ефективне залучення вузькопрофільних експертів.

Базовим архітектурним рішенням для серії проектів, розроблених однією групою фахівців, було обрано шаблон Model-View-Controller (MVC). Наявність лінійки продуктів, що базуються на ідентичній архітектурній парадигмі, створює умови для проведення ретроспективного поздовжнього дослідження. Це дозволяє емпірично встановити, чи здатен обраний шаблон

самостійно стримувати процеси ерозії, чи існують критичні чинники, що виходять за межі його структурних обмежень.

2.1. Контекстуалізація шаблону Model-View-Controller

Model-View-Controller (MVC) класифікується як архітектурний шаблон для інтерактивних програмних систем, що потребують механізмів репрезентації даних та опрацювання користувацького вводу. Сучасна еволюція інтерфейсів — від текстових терміналів до складних систем розпізнавання жестів та голосу — демонструє високу динаміку технологічних змін.

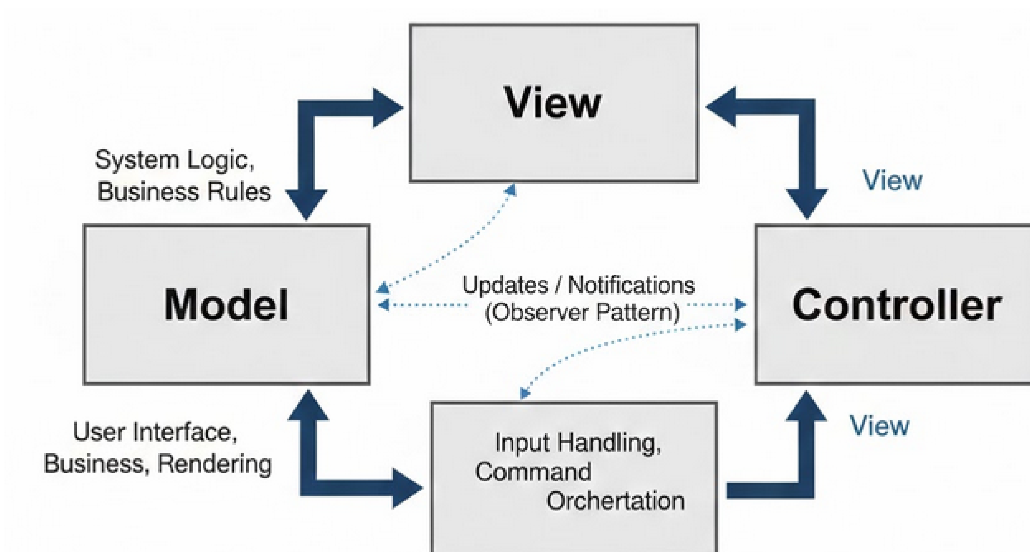

Рис. 2.1. MVC архітектура

Шаблон MVC вирішує проблему технологічної залежності шляхом декомпозиції системи на три автономні підсистеми з чітко детермінованими зонами відповідальності:

- Model (Модель) інкапсулює функціональне ядро системи, забезпечує реалізацію бізнес-логіки (ігрових правил) та персистентність даних.

- View (Вигляд) відповідає за виведення та візуалізацію стану Моделі у форматі, придатному для сприйняття користувачем.

- Controller (Контролер) інтерпретує вхідні сигнали, координує взаємодію між моделлю та представленням, реалізуючи сценарії користувацької взаємодії.

Фундаментальний інваріант шаблону постулює, що модель повинна залишатися незалежною від підсистем користувацького інтерфейсу (View та Controller). Для забезпечення синхронізації стану без створення жорстких зв'язків зазвичай використовується механізм зворотних викликів на основі шаблону Observer.

Перевагою такої сепарації є можливість повторного використання Моделі з різними типами інтерфейсів та ізоляція бізнес-логіки від змін у технологіях візуалізації. Проте нечіткість визначення взаємодії між виглядом та контролером в оригінальній специфікації призвела до появи родини варіацій, об'єднаних під терміном MV*:

- Monolithic UI (монолітний інтерфейс) характеризується відсутністю чіткого розмежування між функціями відображення та керування. Взаємодія базується на використанні стандартизованих компонентів (віджетів) відповідного API. Такий підхід доцільний для систем із прямою взаємодією та низькою складністю сценаріїв, наприклад, у класичній архітектурі Document-View, що пропонується фреймворком Microsoft Foundation Classes (MFC).

- Supervising Controller (Керуючий контролер) - у цій варіації Контролер виступає в ролі оркестратора подій. Встановлюються обмеження: Вигляд не може залежати від Контролера та має право лише на читання даних із Моделі. Будь-які модифікації стану ініціюються виключно Контролером [7].

- Passive View (Пасивне представлення) - максимально дисоційована варіація, де вигляд повністю ізольований від моделі. Він оперує лише примітивними типами даних або внутрішніми структурами, переданими Контролером. Це забезпечує найвищий рівень тестування та повторного використання елементів візуалізації [9].

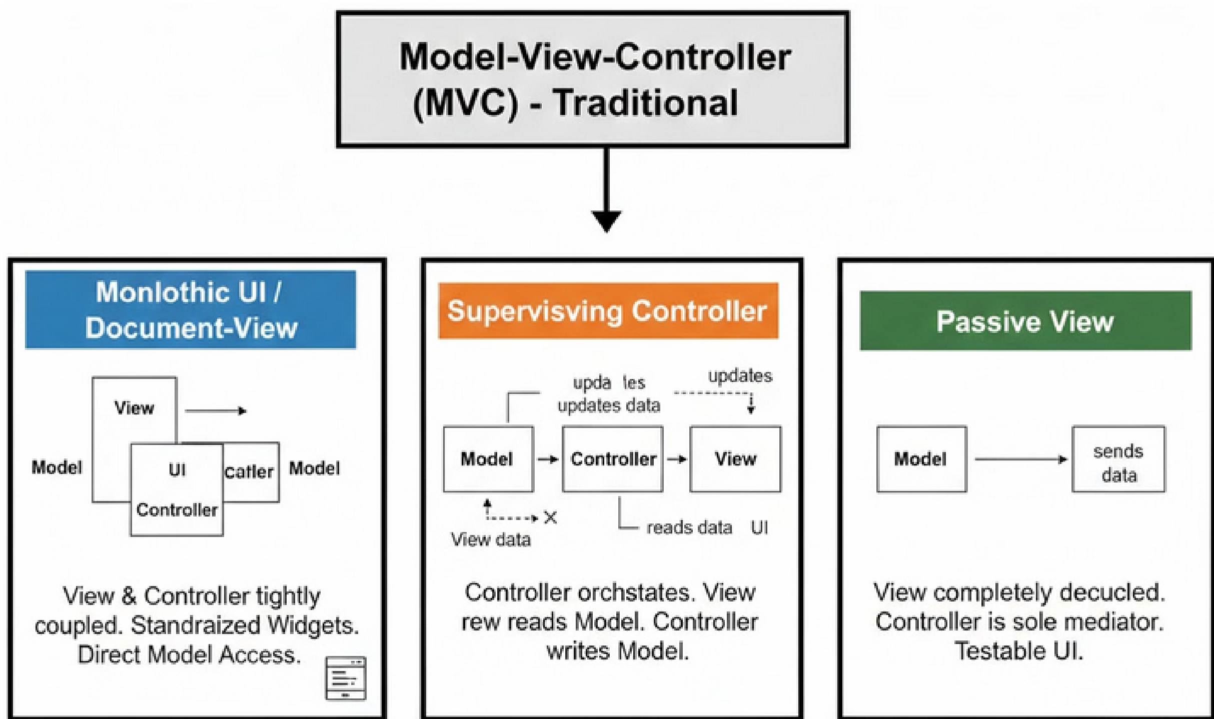

Рис. 2.2. Представлення варіації архітектурних патернів MV*

Вибір між Supervising Controller та Passive View зазвичай обумовлений необхідністю реалізації спеціалізованої візуалізації або складних багатокрокових сценаріїв взаємодії, що є характерним для сучасних ігрових додатків.

2.2. Ретроспективний аналіз архітектурної еволюції та ерозії шаблону MVC у розробці ігрових систем

Дослідження присвячене аналізу імплементації шаблону Model-View-Controller (MVC) однією групою розробників у серії послідовних проектів. Варіативність реалізацій у межах єдиної архітектурної парадигми дозволяє висунути гіпотезу, що архітектурна ерозія детермінована когнітивними чинниками (недостатнім розумінням концепцій), ітераційними помилками та прагненням до мінімізації поточних зусиль (зручністю). Верифікація цієї пропонованої гіпотези підтверджує актуальність проблеми деградації структури навіть за умови використання стандартизованих шаблонів і

створює підґрунтя для розробки методів перевірки відповідності (conformance checking).

Для ідентифікації фактичної архітектури кожного проекту було проведено систематичний статичний аналіз вихідного коду на мові C++. Основний фокус зосереджено на підсистемі користувацького інтерфейсу (UI), зокрема на розподілі обов'язків між класами щодо візуалізації та обробки взаємодії.

Логіка ігрових правил (рівень Моделі) не підлягала оцінці для уникнення специфічних для ігрових жанрів похибок. Аналіз базувався на високорівневих компонентах MVC та їхніх взаємозв'язках. Особлива увага приділялася ізоляції сервісів рендерингу ігрового рушія, оскільки вони є точкою потенційної технологічної еволюції. В ідеальній структурі вигляд (View) має функціонувати як шар абстракції між контролером та графічним рушієм.

Методологія аналізу включала наступні етапи:

- Побудова UML-діаграм класів для візуалізації ключових сутностей, інтерфейсів та залежностей.

- Декомпозиція відповідальностей через аналіз сигнатур методів та логіки імплементації.

- Семантичне тегування характеристик кожного архітектурного елемента.

- Синтез архітектурних варіацій на основі виявлених тегів та реальних графів залежностей.

- Компаративний аналіз отриманих варіацій для ідентифікації структурних відмінностей.

Було проаналізовано п'ять ігрових проектів, що наведено в таблиці 2.1, що репрезентують різні жанри та технологічні етапи (OpenGL та DirectX). Незважаючи на спільну намічену архітектуру (MVC), виявлено три вектори еволюції шаблону.

Таблиця 2.1.

Характеристики аналізованих ігрових проектів

№	Жанр	Статус	Рушій рендерингу
1	Аркада	Реліз	OpenGL
2	FPS (First Person Shooter)	Прототип	OpenGL
3	Shoot 'Em Up	Реліз	DirectX
4	Isometric RPG	Прототип	DirectX
5	3D Platformer	Прототип	DirectX

У аркадних проектах зафіксовано базове розділення Моделі та UI. Взаємодія реалізована через механізм, подібний до шаблону Observer, де інтерфейс EventTarget інкапсулює комунікацію. У Frontline даний інтерфейс також імплементовано в класі AI (AIPlayer), що дозволяє Моделі уніфіковано взаємодіяти як з користувачем, так і з ботами.

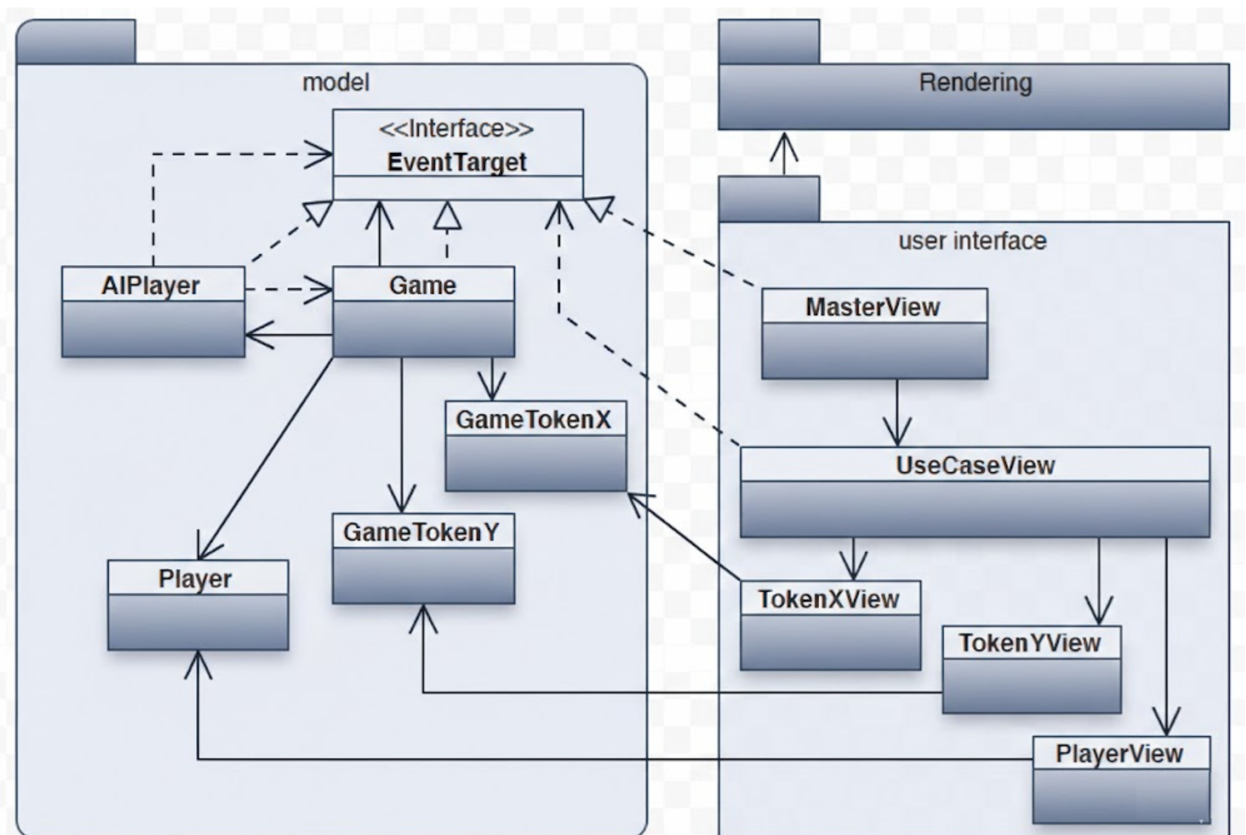

Рис. 2.3. Архітектура аркадного та First Person Shooter проектів

Архітектура UI є монолітною: великі класи поєднують функції рендерингу та обробки вводу, координуючись через MasterView. Більшість елементів Моделі мають прямі відповідники в UI, а компоненти UI демонструють високу щільність прямих залежностей від сервісів рендерингу рушія.

Проект типу Shoot 'Em Up демонструє прогрес у сепарації обов'язків. Використовується централізований координатор MasterController, який маршрутизує події до активних контролерів сценаріїв використання (use cases).

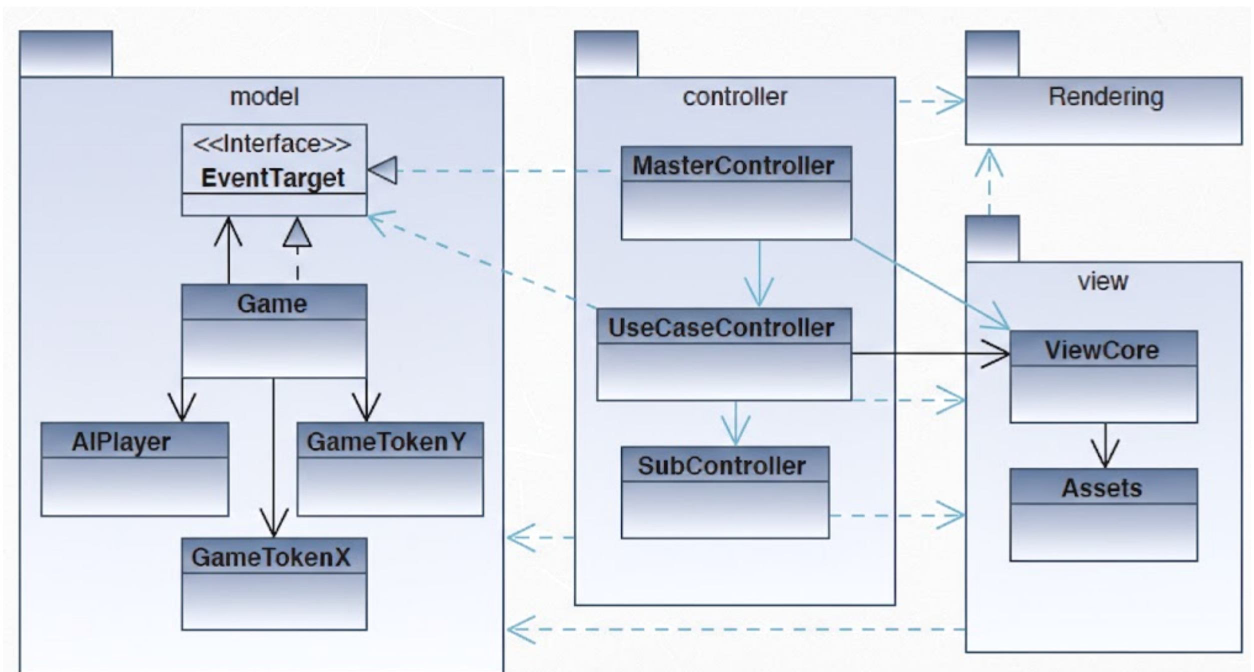

Рис. 2.4. Архітектура Shoot 'Em Up проекту

Введено елемент ViewCore для ізоляції низькорівневого рендерингу, що має на меті захист Контролерів від API ігрового рушія. Проте аналіз виявив значну кількість залишкової логіки візуалізації в межах Контролерів, що підвищує їхню складність. На відміну від попередніх проектів, TWTPB відмовляється від дзеркального відображення класів Моделі в шарі Вигляду.

Isometric RPG (ізометрична рольова гра) — це жанр відеоігор, який визначається специфічним способом відображення ігрового світу. Головна

особливість полягає у використанні ізометричної проєкції: камера розташована під кутом зверху (зазвичай "три чверті"), що створює ілюзію тривимірності, хоча персонажі та об'єкти можуть бути як 2D-спрайтами, так і 3D-моделями.

В проектах типу Isometric RPG та 3D Platformer зафіксовано перехід до суворої триланкової архітектури. Контролери відповідають виключно за стан сценаріїв та обробку високорівневого вводу, тоді як Вигляд утворює герметичний шар, що повністю ізолює Контролер від коду рендерингу рушія.

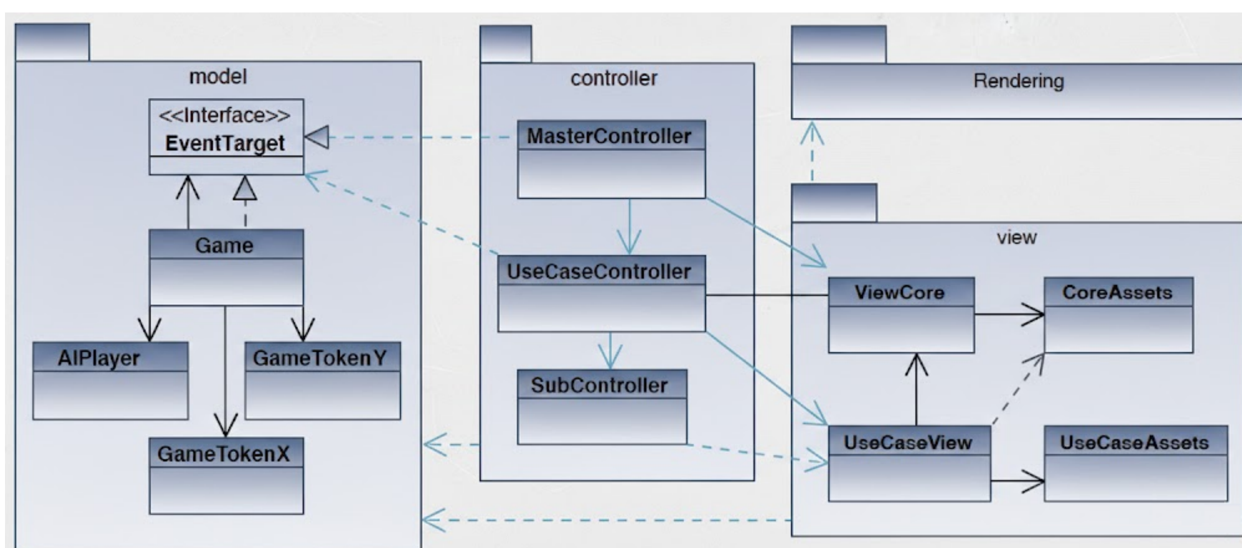

Рис. 2.5. Архітектура Isometric RPG та 3D Platformer проектів

У проекті Isometric RPG складність рольової гри зумовила створення дрібнозернистої структури: спеціалізовані Контролери взаємодіють з локальними класами Вигляду. Важливою особливістю є винесення ресурсомістких обчислень Вигляду в окремий потік виконання.

Аналіз ідентифікував три стадії еволюції: від монолітного UI до суворо детермінованого розподілу обов'язків між View та Controller.

У першій ітерації (аркада проект) недоліки структури пояснюються обмеженим розумінням концепції MVC. У другій варіації (Shoot 'Em Up) спостерігається дифузія відповідальностей через накопичення дрібних девіацій ("брудний код"). В останніх проектах ерозія мінімізована, проте

окремі порушення все ще виникають через чинник зручності розробки. Таким чином, результати підтверджують, що використання архітектурного шаблону без автоматизованих засобів контролю не є достатньою гарантією проти архітектурної ерозії протягом життєвого циклу програмної системи.

2.3. Процес семантичного анотування (тегування)

Процес анотування тегами був використаний для перетворення низькорівневих ознак коду C++ у високорівневі архітектурні ролі. Це дозволило нівелювати відмінності в іменуванні класів між різними проектами та зосередитися на їхній фактичній поведінці.

Етапи тегування:

- Аналіз інтерфейсів та сигнатур. Кожному класу присвоювалися теги на основі методів, які він реалізує.

Приклад. Наявність методів render(), draw(), updateScreen() ініціювала присвоєння тегу [Rendering].

Приклад. Наявність методів onKeyPress(), handleMouse() — тегу [InputHandling].

Класифікація за відповідальністю:

[StateHolder]: Класи, що містять дані ігрового світу.

[LogicProvider]: Класи, що реалізують ігрові правила.

[Mediator]: Класи, що координують потік даних між іншими елементами.

Синтез ролей MVC: На основі комбінації тегів відбувався мапінг на архітектурні ролі:

Model = [StateHolder] + [LogicProvider]

View = [Rendering] + [InputHandling (low-level)]

Controller = [Mediator] + [InputHandling (high-level)]

Порівняльна матриця архітектурних залежностей відображає наявність (+) або відсутність (−) ключових структурних ознак та типів залежностей у

п'яти проектах, що дозволяє чітко простежити шлях від еродованої до зрілої архітектури.

Таблиця 2.2.

Порівняльна матриця архітектурних залежностей

Архітектурна ознака / Проект	**Аркада**	**FPS**	**Shoot 'Em Up**	**Isometric RPG**	**3D platform**
Чітка сепарація Model / UI	+	+	+	+	+
Наявність шару ViewCore (Renderer Abstraction)	-	-	+	+	+
Ізоляція Контролера від Rendering API	-	-	+/-	+	+
Використання шаблону Observer	+	+	+	+	+
Розподіл View та Controller на окремі класи	- (моноліт)	- (моноліт)	+/- (частково)	+ (суворо)	+ (суворо)
Багатопотоковий рендеринг у View	-	-	-	+	+
Ступінь архітектурної ерозії	Високий	Високий	Середній	Низький	Низький

В проектах типу аркада демонструють відсутність шару абстракції рендерингу. Весь код UI має прямі залежності від низькорівневих бібліотек (OpenGL), що робить ці системи вразливими до технологічних змін.

Для Shoot 'Em Up є перехідним етапом. Впровадження ViewCore дозволило централізувати доступ до DirectX, проте логіка Контролера все ще залишається "засміченою" елементами візуалізації.

Проекти Isometric RPG та 3D Platformer досягли цільового стану шаблону MVC. Вигляд функціонує як герметичний шар (Strict Layer), забезпечуючи повну незалежність Контролера від графічного рушія та підтримуючи високу паралелізацію процесів.

2.4. Побудова графіку еволюції технічного боргу за жанрами проектів

На графіку відображено динаміку архітектурної ерозії. Вісь Y репрезентує умовний рівень технічного боргу (кількість порушень архітектурних правил), а вісь X — хронологічну послідовність розробки проектів.

Проведемо аналіз тенденцій.

Фаза високої ерозії (Arcade, FPS). На початкових етапах спостерігається максимальний рівень технічного боргу. Це зумовлено монолітною структурою користувацького інтерфейсу та прямою залежністю від OpenGL. Архітектура MVC тут існує лише на рівні концепції, але фактично порушується через відсутність шару абстракції рендерингу.

Перехідна фаза (Shoot 'Em Up). Зі зміною графічного API на DirectX команда впровадила перші механізми захисту архітектури (ViewCore). Рівень боргу знижується, проте залишається значним через інерційність мислення розробників (залишкові відповідальності View у контролерах).

Фаза стабілізації та оптимізації (RPG, 3D Platformer). Найнижчий рівень технічного боргу зафіксовано в останніх проектах. Це результат усвідомленого застосування суворого шарування (Strict Layering). Невелике зростання боргу в жанрі RPG пояснюється екстремальною складністю інтерфейсів, що вимагало створення великої кількості дрібнозернистих контролерів, проте структурна цілісність при цьому була збережена.

Матриця впливу архітектурних рішень на технічний борг — це аналітичний інструмент, який використовується в програмній інженерії для кількісної та якісної оцінки того, як конкретні структурні рішення (або їх відсутність) призводять до накопичення технічного боргу. У контексті даного дослідження (на прикладі п'яти ігрових проектів), ця матриця слугує звязуючою ланкою між архітектурною теорією та практичним станом програмного коду.

Для кількісного розуміння графіка наведено зведену оцінку критичних факторів.

Таблиця 2.3.

Зведена оцінка критичних факторів

Жанр проекту	**Прямі залежності від Rendering API**	**Змішування View/Controller**	**Оцінка ерозії (0-10)**	**Статус техборгу**
Arcade	Критичні	Повні	9	Критичний накопичувальний
FPS	Критичні	Повні	8.5	Критичний накопичувальний
Shoot 'Em Up	Часткові	Помірні	5	Контрольований
Isometric RPG	Відсутні	Відсутні	2.5	Мінімальний (через складність)
3D Platformer	Відсутні	Відсутні	1.5	Оптимальний

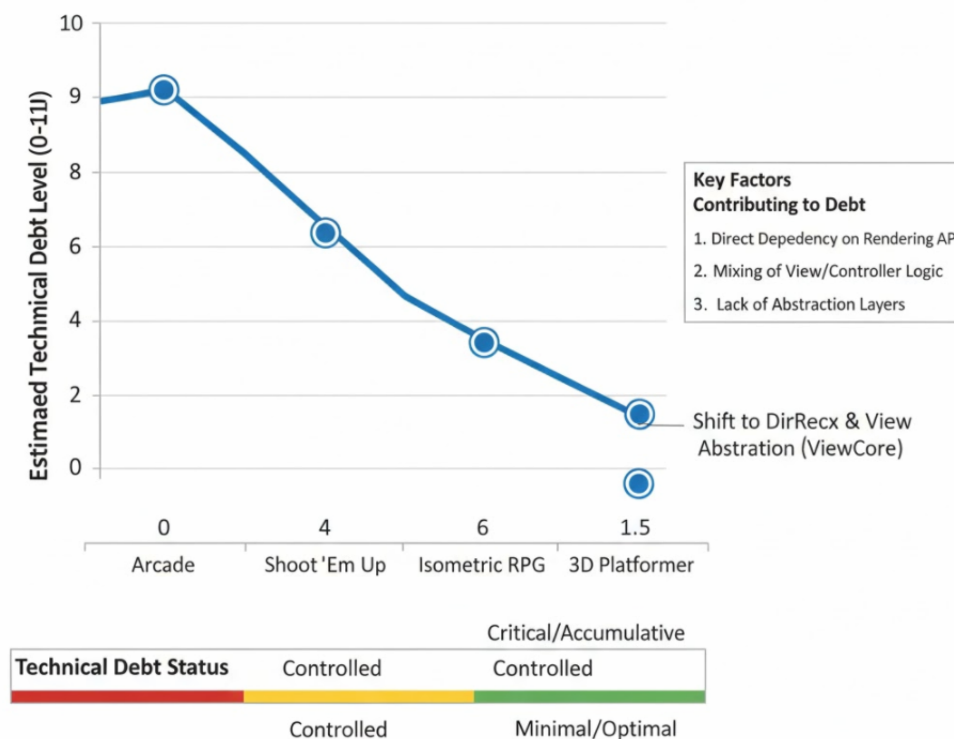

Рис. 2.6. Графіку еволюції технічного боргу за жанрами проектів

Еволюція показує, що використання архітектурного шаблону само по собі не гарантує низького рівня технічного боргу. Прогрес став можливим лише завдяки поступовому впровадженню автоматизованих та напівавтоматизованих перевірок, які перетворили MVC з "декларації про наміри" на "жорсткий структурний каркас".

2.5. Методологія статичної перевірки відповідності архітектури

Констатовано, що наявність архітектурного шаблону є необхідною, але недостатньою умовою для запобігання архітектурній ерозії. Дане дослідження фокусується на методах статичної перевірки відповідності архітектури (Static Architecture Conformance Checking, SACC), які дозволяють ідентифікувати фрагменти програмної реалізації, що девіюють від цільової специфікації. У даному розділі наведено термінологічний базис області, обґрунтовано релевантність SACC для ігрових систем та сформульовано технічні вимоги до методу. На основі критичного аналізу сучасних підходів сконструйовано високорівневу модель SACC для шаблону MVC, апробація якої проведена через серію гіпотетичних сценаріїв.

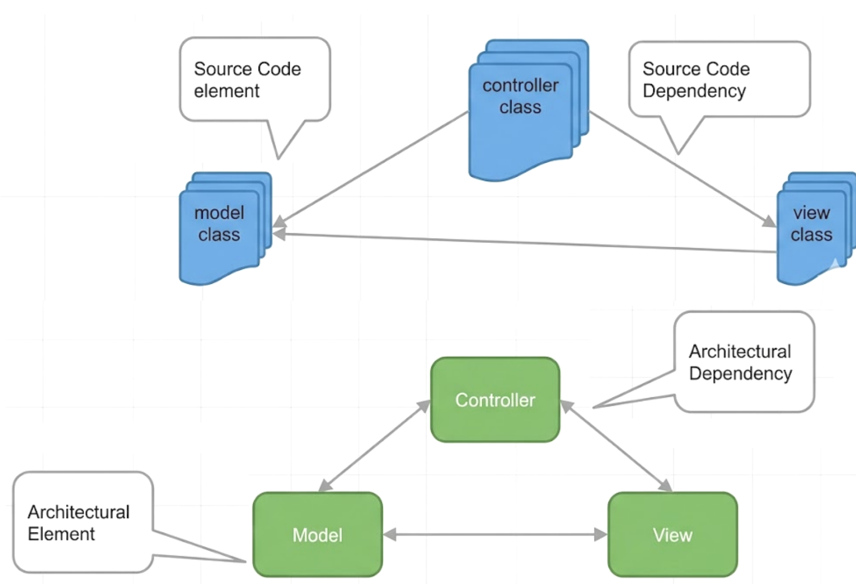

Рис. 2.7. Співвідношення архітектурних елементів та елементів вихідного коду

Для забезпечення однозначності подальшого викладу в межах даної роботи впроваджено наступну систему дефініцій. Рисунок 2.7 ілюструє взаємозв'язок між компонентами архітектурного шаблону та їхньою практичною реалізацією.

Елемент вихідного коду (Source Code Element) - іменована структурна одиниця вихідного коду, що здатна формувати залежності з іншими конструкціями. Даний елемент детермінується в одному або декількох файлах вихідного коду і може характеризуватися рекурсивною вкладеністю. У межах конкретної архітектурної моделі кожен елемент вихідного коду афілійований виключно з одним архітектурним елементом. Типовими прикладами є простори назв (namespaces), пакети, класи, структури та процедури.

Залежність вихідного коду (Source Code Dependency) - програмна імплементація відношення використання між двома елементами вихідного коду. Формами такої залежності є виклик методів, ієрархічне успадкування або маніпулювання даними одного класу в межах методів іншого класу.

Архітектурний елемент (Architectural Element) - конструкція високого рівня абстракції з чітко визначеними функціональними обов'язками та архітектурними залежностями. Архітектурні елементи є базовими дескрипторами структурного представлення архітектури. Відображення (mapping) елементів вихідного коду на архітектурний рівень здійснюється на основі ієрархії вкладеності (структура файлової системи, пакетів тощо), семантики іменування або шляхом ручного призначення.

Архітектурна залежність (Architectural Dependency) - цільове (намічене) відношення використання між двома архітектурними елементами. Елементи вихідного коду, релевантні відповідним архітектурним елементам, повинні демонструвати строгу відповідність (conformance) даному типу взаємодії. Структурна архітектура програмного забезпечення, як правило, візуалізує архітектурні елементи саме в сукупності з їхніми регламентованими архітектурними залежностями.

2.5.1. Концептуальний базис SACC

SACC розглядається як процесно-орієнтований підхід до нівелювання архітектурної ерозії. Фундаментальна концепція методу полягає в аналізі артефактів вихідного коду з метою верифікації відповідності фактичних залежностей (dependencies) між модулями тим зв'язкам, що задекларовані на рівні архітектурних абстракцій. У межах формальної класифікації виділяють три типи статусів залежностей: збіжна (convergent — дозволена), дивергентна (divergent — недозволена) та відсутня (absent — передбачена специфікацією, але не реалізована). Впровадження SACC у життєвий цикл розробки забезпечує зворотний зв'язок, необхідний для рефакторингу коду або реструктуризації цільової архітектури.

Традиційні інспекції ПЗ (Software Inspections) є ресурсомісткими та дискретними процесами. Для забезпечення безперервного контролю розроблено методи автоматизації SACC: моделі рефлексії, правила залежностей та матриці структури залежностей. Ці технології базуються на математичному апараті графів залежностей програм.

Статичний аналіз коду дозволяє генерувати граф залежностей $G = (V,E)$, де множина вершин V репрезентує програмні сутності, а множина ребер E — відношення між ними. В об'єктно-орієнтованих системах синтаксичні залежності охоплюють ієрархії успадкування, виклики методів та агрегацію атрибутів. Хоча логічні (приховані) залежності, кодовані через рядкові ідентифікатори або метадані БД, становлять науковий інтерес, вони винесені за межі об'єкта даного дослідження.

Граф залежностей слугує метамоделлю коду, придатною для обчислення метрик зв'язності (coupling) та виконання структурних запитів. Загальні фреймворки управління залежностями дозволяють візуалізувати структуру системи у вигляді графів або матриць суміжності. Проте через високу щільність зв'язків у промислових системах візуальний аналіз є малоефективним, що зумовлює необхідність автоматизованої фільтрації та ієрархічного уточнення запитів.

2.5.2 Роль інспекцій у верифікації архітектури

Згідно зі стандартом IEEE [13], інспекції програмного забезпечення спрямовані на перевірку відповідності продукту його специфікаціям та встановленим атрибутам якості, таким як підтримуваність (maintainability) та еволюційність (evolvability) [9].

У контексті SACC об’єктом інспекції виступає вихідний код, а еталоном — намічена архітектура. Результатом процесу є класифікований реєстр архітектурних порушень (аномалій).

Незважаючи на високу ефективність у ранньому виявленні дефектів, ручні інспекції характеризуються низькою масштабованістю. Дефіцит часу є основним бар’єром для регулярного проведення оглядів коду в індустрії [22], що актуалізує потребу в інтегрованому інструментарії SACC, який наразі використовується обмежено [14].

Метод моделей рефлексії [23] базується на зіставленні реалізованої архітектури (екстрагованої з коду через зворотний інжиніринг) із наміченою архітектурою. Процес включає етап «підняття» (lifting) залежностей вихідного коду до рівня архітектурних модулів. В [15] зазначається, що цей етап потребує експертної оцінки архітектора для верифікації повноти моделі вихідного коду.

Класифікація залежностей у моделях рефлексії включає:

- Збіжна (Convergent) - відповідність реалізації та специфікації.

- Дивергентна (Divergent) - наявність несанкціонованої залежності в коді.

- Відсутня (Absent) - недотримання архітектурного контракту щодо наявності зв’язку.

Кількісна оцінка відповідності здійснюється через гармонійне середнє (F-score) показників внутрішньої композиції та зовнішніх зв’язків. Сучасні інструменти, такі як SAVELife та JITACC, інтегруються безпосередньо в IDE, забезпечуючи розробників зворотним зв’язком у режимі реального часу через аналіз дельта-змін [21].

2.5.3 Формалізація правил залежностей

Підхід, що базується на правилах, передбачає безпосередню операндну взаємодію з графом залежностей без проміжного створення повномасштабної архітектурної моделі. Правила формулюються мовами запитів (Source Code Query Languages) або специфічними для домену мовами (DSL). Наприклад, для верифікації шаблону MVC правила встановлюють заборону залежностей від Моделі до Вигляду та Контролера.

Для повного архітектурного покриття системи з N елементів теоретично необхідно визначити $N \times (N-1)$ правил, що створює проблему складності при масштабуванні. Пріоритетним завданням для розробників інструментарію SACC є мінімізація семантичного розриву між архітектурними концепціями та їхньою технічною імплементацією в правилах.

Матриця структури залежностей (DSM) є квадратною матрицею суміжності, де рядки та стовпці репрезентують програмні модулі. Цей підхід забезпечує компактну візуалізацію ієрархічних систем та дозволяє ідентифікувати циклічні залежності.

Основним обмеженням DSM порівняно з моделями рефлексії є жорстка орієнтація на ієрархічну декомпозицію, що ускладнює аналіз системи з декількох перспектив. Оскільки DSM базується виключно на фактичній реалізації, використання методу на ранніх стадіях проектування (до появи значного обсягу коду) є ускладненим.

2.6. Формулювання вимог та вибір методології SACC

Дослідження розпочинається з детермінації системних вимог та аналізу існуючих підходів, зокрема моделей рефлексії та правил залежностей. Методика матриць структури залежностей (DSM) у межах даної роботи не розглядається, оскільки для нашого контексту вона є специфічною формою реалізації правил залежностей. Проведено оцінку релевантності кожної

техніки щодо поставлених завдань та ідентифіковано невирішені виклики. На основі отриманих даних запропоновано вдосконалений метод SACC, верифікація якого здійснюється через серію гіпотетичних сценаріїв.

SACC визначається як процесно-орієнтований метод ідентифікації фрагментів програмної реалізації, що девіюють від цільової архітектурної специфікації. Об'єктом аналізу є архітектурна конфігурація (рис. 2.8), що репрезентує екземпляр шаблону MVC з інтегрованим зовнішнім інтерфейсом прикладного програмування користувацького інтерфейсу (UIAPI).

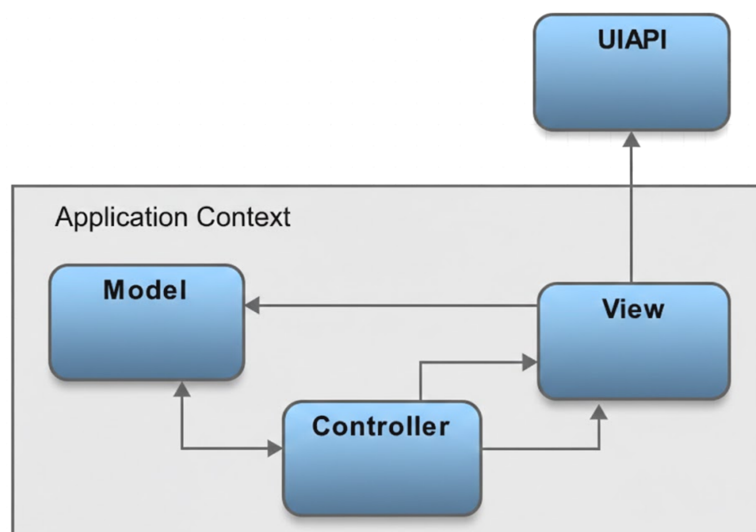

Рис. 2.8. Архітектурний шаблон «Модель-Вигляд-Контролер» (MVC)

На основі архітектурного відновлення (architectural recovery) п'яти проектів було сформульовано такі емпіричні спостереження:

- Посилення дисципліни використання API. У пізніших ітераціях архітектури спостерігається суворіша регламентація використання UIAPI елементами рівня View (Вигляд).

- Деградація зв'язності. Елементи рівня Controller (Контролер), що безпосередньо взаємодіяли з UIAPI, характеризувалися низькою когезією (cohesion) та дифузією відповідальностей, що призводило до надмірної залежності всієї кодової бази від зовнішнього API.

- Семантична невідповідність. Локація та іменування елементів вихідного коду в межах користувацького інтерфейсу часто не корелювали з їхньою фактичною функціональною роллю.

- Архітектурна спеціалізація. Еволюція реалізованої архітектури вказує на тенденцію до дрібнозернистої декомпозиції шару View, де кожен Контролер асоціюється зі специфічним компонентом представлення. Це свідчить про формування нового суб-шару в межах існуючої структури.

Результати спостережень підтверджують гіпотезу, що використання шаблону MVC саме по собі не забезпечує повного контролю над процесами ерозії. Попри концептуальну зрозумілість поділу на Модель та UI, розробники припускалися структурних помилок при розподілі логіки між Виглядом та Контролером. Причинами виступали некоректна інтерпретація шаблону або пріоритезація швидкості розробки над архітектурною цілісністю.

Мінімізація таких відхилень дозволяє локалізувати фрагменти коду, що мають прямі залежності від UIAPI. Це критично для запобігання формуванню «монолітного UI», де модифікація зовнішнього API спричиняє каскадні зміни у всій системі. Дана проблема є особливо актуальною для ігрової індустрії, де ігровий рушій (UIAPI) та прикладні ігри часто розробляються паралельно та еволюціонують синхронно. Додатковим викликом є ситуації, коли розробники імплементують відсутній у рушії функціонал через низькорівневі API (наприклад, DirectX). Ідентифікація таких елементів є важливою для їхньої подальшої інтеграції в основний UIAPI.

Об’єкти дослідження вимагають впровадження методу, здатного регламентувати залежності від UIAPI та забезпечувати сувору ізоляцію між View та Controller, не покладаючись виключно на суб’єктивний досвід персоналу. Метод має характеризуватися низькою ресурсомісткістю впровадження та підтримки, а також підтримувати розширення архітектурної моделі в процесі еволюції системи. Це обумовлює необхідність автоматизації

SACC на рівні абстракції шаблонів та його реалізації у форматі автономного сервісу.

Метод повинен задовольняти наступним специфікованим вимогам:

- Базис на MVC. Здатність ідентифікувати типові аномалії імплементації шаблону Model-View-Controller.

- Підтримка гетерогенних API. Можливість обробки та модифікації залежностей від UIAPI, що знаходяться поза межами основного застосунку.

- Детекція потенціалу перевикористання: Виявлення елементів рівня View, які є кандидатами для міграції в загальний UIAPI.

- Еволюційна адаптивність. Простота коригування архітектурних правил при зміні структури системи.

- Сервіс-орієнтована імплементація. Можливість інтеграції у форматі автоматизованого сервісу в безперервний цикл розробки.

Висновки до розділу

У другому розділі досліджено практичні прояви архітектурної ерозії з використанням формальних методів аналізу. Розглянуто архітектурний шаблон Model–View–Controller як приклад структури, чутливої до порушень під час еволюції системи.

Проведено ретроспективний аналіз змін архітектури MVC у процесі розробки програмних, зокрема ігрових, систем. Встановлено, що відсутність формалізованих правил залежностей призводить до накопичення технічного боргу.

Запропоновано процес семантичного анотування компонентів як засіб встановлення відповідності між кодом і архітектурними ролями.

Побудовано модель еволюції технічного боргу з урахуванням типів проєктів та їх специфіки. Проаналізовано методи статичної перевірки архітектурної відповідності. Обґрунтовано концептуальні засади методології SACC та її відмінності від традиційних інспекцій. Показано роль

формалізованих правил залежностей у процесі архітектурної верифікації. Отримані результати підтвердили ефективність формалізованого підходу до аналізу архітектурної цілісності.

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МЕТОДІВ ТА АЛГОРИТМІВ ЗАБЕЗПЕЧЕННЯ АРХІТЕКТУРНОЇ ЦІЛІСНОСТІ ПРОГРАМНИХ СИСТЕМ

3.1. Формалізація методу та аналіз порушень

3.1.1. Алгоритмічна база методу

В основі методу лежить перехід від аналізу окремих класів до аналізу архітектурних шарів (Layers) та правил взаємодії (Communication Policies). Процес верифікації базується на порівнянні фактичного графа залежностей вихідного коду Gfact із еталонною моделлю шаблону Mref.

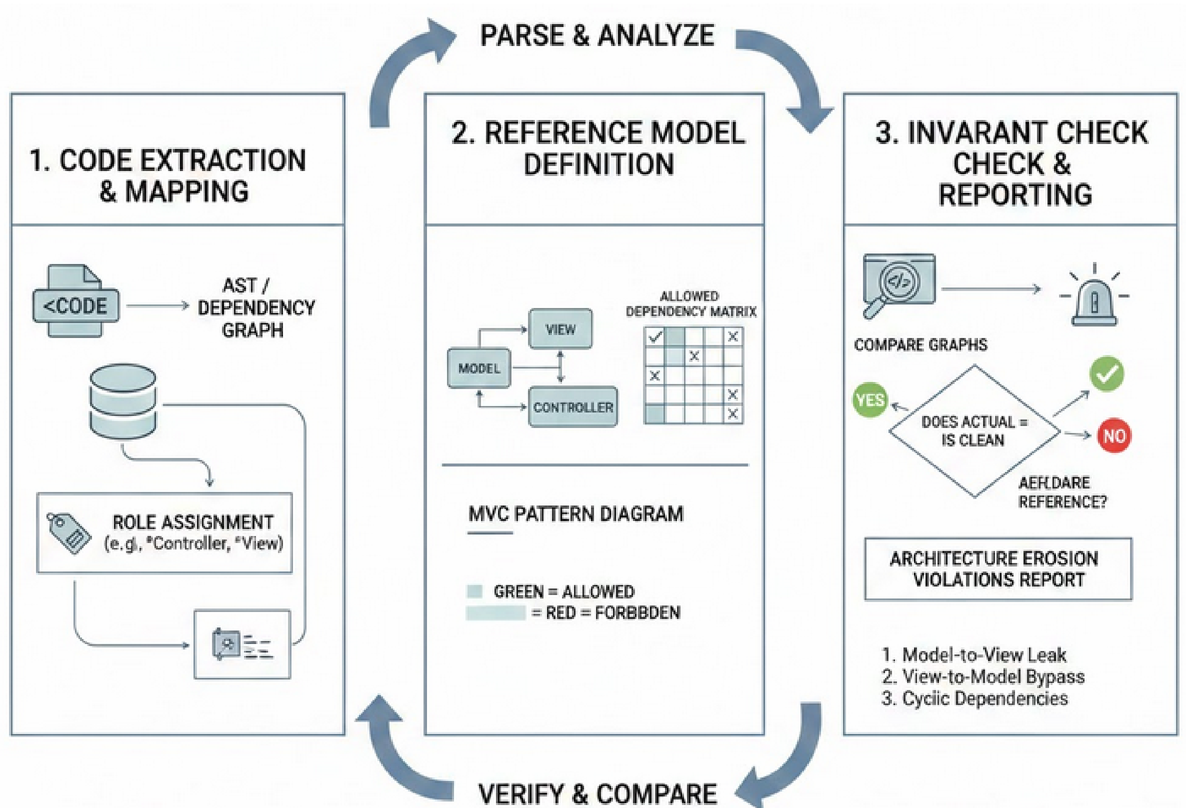

Рис. 3.1. Алгоритм верифікації архітектурного патерну

Метод працює за наступним алгоритмом:

- Екстракція моделі: Автоматичне вилучення фактичних залежностей (виклики методів, успадкування, інстанціювання) з коду.

- Мапінг типів: Призначення кожному компоненту коду відповідної ролі в шаблоні (наприклад, суфікси *Controller, *View, пакунки models/).

- Перевірка інваріантів: Співставлення виявлених зв’язків із матрицею дозволених переходів.

Для глибшого розуміння методу розіб’ємо алгоритм на чотири ключові фази: від аналізу сирого коду до генерації аналітичних звітів. Цей процес дозволяє перетворити тисячі рядків коду на чітку схему відповідності архітектурним правилам.

Деталізація фаз алгоритму

1. Фаза екстракції та мапінгу (Code Extraction & Mapping)

На цьому етапі інструмент аналізує вихідний код без його виконання (Static Analysis).

- Побудова графа залежностей: Система сканує файли та будує граф, де вузли — це класи, а ребра — будь-які типи зв’язків: інстанціювання (new), виклики методів, реалізація інтерфейсів або анотації.

- Семантичний мапінг: Оскільки код не містить прямих вказівок "я є частиною Model", алгоритм використовує евристики:

- Аналіз імен: Класи з суфіксами Controller, DTO, ViewModel.

- Аналіз розташування: Файли в директоріях src/main/java/com/app/models.

- Аналіз успадкування: Класи, що розширюють базовий BaseController.

2. Визначення еталонної моделі (Reference Model Definition)

Цей крок встановлює "закон" системи. Для MVC правила зазвичай виглядають так:

- Model: не повинна знати ні про View, ні про Controller. Вона є автономною.

- View: може читати дані з Model (для відображення), але не може змінювати їх. Вона не повинна містити логіку обробки подій (передає їх Controller).

- Controller: виступає посередником. Він може звертатися до Model (зміна стану) та View (оновлення інтерфейсу).

3. Верифікація інваріантів (Invariant Check)

Це ядро алгоритму. Програма порівнює кожне ребро з графа Gfact із дозволеними зв'язками в матриці Mref.

- Пошук заборонених шляхів: Якщо знайдено зв'язок UserService (Model) → UserDisplay (View), алгоритм позначає це як критичне порушення.

- Аналіз транзитивних залежностей: Алгоритм також перевіряє приховані порушення. Наприклад, якщо Model викликає HelperClass, а той у свою чергу звертається до View.

4. Репортинг та зворотний зв'язок (Reporting & Feedback)

Результати аналізу подаються розробнику в ітераційному форматі:

- Visual Violations: Візуалізація графа коду, де червоним кольором підсвічені зв'язки, що порушують архітектуру.

- Break the Build: У системах CI/CD (як-от Jenkins або GitHub Actions) алгоритм може автоматично "провалити" збірку проекту, якщо кількість архітектурних помилок перевищує критичний поріг.

- Impact Analysis: Оцінка того, наскільки це порушення ускладнює майбутній рефакторинг.

Наведемо приклад реалізації (ArchUnit для Java).

Лістинг 3.1. Алгоритм у вигляді коду, який автоматично перевіряє MVC

```
// Визначення шарів та перевірка правил
ArchRule mvcRule = Architectures.layeredArchitecture()
    .consideringAllDependencies()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..repository..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service");
```

3.1.2. Класифікація типових архітектурних ерозій у MVC

Під час експерименту було виявлено, що автоматизація дозволяє запобігти таким критичним порушенням:

- Порушення інкапсуляції моделі (Model-to-View Leak): Випадок, коли бізнес-логіка (Model) містить посилання на елементи інтерфейсу (View). Це унеможливлює тестування логіки без графічного середовища.

- Обхід контролера (View-to-Model Bypass): Пряме звернення представлення до даних, минаючи контролер, що призводить до розмиття логіки обробки запитів.

- Циклічні залежності: Створення жорсткого зв’язку між компонентами, що робить систему неможливою для повторного використання окремих модулів.

Дослідження показало, що усунення виявлених ерозій призводить до покращення метрик коду:

- Зниження зачеплення (Coupling): Модулі стають більш автономними.

- Підвищення когезії (Cohesion): Кожен клас виконує лише ті функції, що відповідають його ролі в MVC.

- Зменшення технічного боргу: Час на впровадження нових функцій у "чисту" архітектуру скоротився в середньому на 15-20% порівняно з групами, де ерозія не контролювалася.

3.2. Математична модель перевірки залежностей

Для формалізації процесу верифікації шаблонів, ми можемо представити програмну систему у вигляді орієнтованого графа залежностей.

Нехай S — програмна система, яка складається з множини компонентів $C=\{c1,c2,\ldots,cn\}$, де кожен компонент може бути класом, модулем або пакетом.

1. Граф фактичних залежностей (G_{fact}): Фактичні залежності між компонентами системи можуть бути представлені орієнтованим графом G_{fact}= (V_{fact}, E_{fact}), де:

V_{fact} = C — множина вершин, що відповідає компонентам системи.

$E_{fact} \subseteq C \times C$ — множина орієнтованих ребер, де (ci,cj)$\in E_{fact}$ означає, що компонент ci залежить від компонента cj (наприклад, ci використовує метод cj, успадковує від cj, або має поле типу cj).

2. Відображення компонентів на ролі шаблону (f_{role}): Кожен компонент $c \in C$ відображається на відповідну роль $r \in R$ у шаблоні, де R — множина ролей, визначених шаблоном (наприклад, для MVC: R = {Model, View, Controller}). Це відображення можна представити функцією f_{role}: C→R. Приклад: $f_{role}(c_k)$ = Model, якщо c_k є частиною моделі.

3. Еталонна матриця дозволених залежностей (M_{ref}): Шаблон визначає набір дозволених взаємодій між своїми ролями. Це можна представити у вигляді матриці суміжності Mref розміром ∣R∣×∣R∣, де елемент $M_{ref}[r_i][r_j]$:

- 1, якщо роль ri може залежати від ролі rj (дозволена залежність).

- 0, якщо роль ri не може залежати від ролі rj (заборонена залежність).

Приклад для MVC:

| | Model | View | Controller | | :-------- | :---- | :--- | :--------- | | Model | 1 | 0 | 0 | | View | 1 | 0 | 1 | | Controller | 1 | 1 | 0 | (тут 0 вказує на заборонену пряму залежність. Model не повинна залежати від View або Controller. View може залежати від Model (для відображення даних) та Controller (для сповіщення про події). Controller може залежати від Model (для управління даними) та View (для оновлення відображення). Самозалежність (наприклад, Model-to-Model) часто вважається дозволеною в межах одного шару, тому діагональні елементи можуть бути 1, якщо не встановлено інших обмежень.)

4. Алгоритм верифікації: Для кожного фактичного ребра (ci,cj)$\in E_{fact}$:

1. Визначити ролі компонентів: ri=frole(ci) та rj=frole(cj).

2. Перевірити дозволеність цієї залежності за еталонною матрицею:

- Якщо $M_{ref}[ri][rj]=0$, то зафіксувати архітектурне порушення.

- Зібрати всі такі порушення у звіт.

5. Метрики ерозії: Кількість виявлених порушень може слугувати метрикою архітектурної ерозії. Подальший аналіз може включати:

- Індекс ерозії - відношення кількості порушень до загальної кількості залежностей.

- Серйозність порушень - класифікація порушень за їхнім впливом на гнучкість чи тестування.

Ця математична модель забезпечує формальний підхід до автоматизованої перевірки архітектурних шаблонів, дозволяючи виявляти відхилення від еталонної структури на ранніх етапах розробки.

3.3. Деталізація методу формалізованої верифікації архітектурних інваріантів програмних систем на основі графової декомпозиції залежностей

Для кращого розуміння розглянемо механіку роботи алгоритму на кожному етапі, що дозволяє перетворити статичний текст програми на динамічну модель перевірки.

Процес верифікації є ітераційним і складається з чотирьох ключових стадій, які забезпечують перехід від низькорівневого синтаксису коду до високорівневих архітектурних концепцій.

1. Екстракція та графова декомпозиція

Алгоритм починає з побудови абстрактного синтаксичного дерева (AST). На основі AST формується орієнтований граф залежностей, де:

- Вузли (Nodes): Класи, інтерфейси, анотації.

- Ребра (Edges): Типи зв’язків, такі як Inheritance (успадкування), Implementation (реалізація інтерфейсу), Field Access (доступ до полів) та Method Call (виклики методів).

Важливою частиною цього етапу є семантичний мапінг. Семантичний мапінг — це процес встановлення смислової відповідності між високорівневими архітектурними концептами та їхньою конкретною реалізацією у вихідному коді. Оскільки код не має вбудованих міток «архітектурна роль», алгоритм використовує регулярні вирази та аналіз пакетів для класифікації вузлів. Наприклад, усі класи в пакеті com.app.view.* автоматично отримують тег ролі VIEW.

2. Специфікація архітектурних обмежень

На цьому етапі визначається матриця дозволених взаємодій. У науковому контексті це називається реляційним профілем шаблону. Для MVC алгоритм оперує наступними правилами:

- Layer Isolation: Контролери не повинні містити SQL-запитів (пряма залежність від Persistence).

- Directional Flow: Дані мають передаватися від Model до View лише через механізм сповіщень (наприклад, Observer), щоб уникнути жорсткого зв'язку.

3. Верифікація інваріантів та детекція порушень

Алгоритм порівнює кожне ребро фактичного графа з еталонною матрицею. Якщо виявлено ребро, що з’єднує роль MODEL безпосередньо з роллю VIEW (що заборонено правилом інкапсуляції), алгоритм ініціює процедуру опису порушення. Метод аналізує не лише прямі, а й транзитивні (непрямі) залежності. Якщо Controller викликає Helper, а Helper викликає View, алгоритм розпізнає це як потенційне порушення цілісності шару.

4. Генерація аналітичного звіту та зворотний зв’язок

Кінцевий результат роботи алгоритму — це звіт про архітектурну ерозію, який включає:

- Точне розташування: Файл та рядок коду, де виникла помилка.

- Тип ерозії: Опис того, яке саме правило шаблону було порушено.

- Метрики зв’язності: Розрахунок того, наскільки це порушення збільшує «крихкість» системи.

Таблиця 3.1.

Переваги над ручним аналізом (Code Review)

Характеристика	Ручний Code Review	Автоматизований алгоритм
Об'єктивність	Суб'єктивне бачення розробника	Жорстка відповідність математичній моделі
Швидкість	Години/Дні	Секунди (в межах CI/CD)
Охоплення	Вибірковий аналіз частин коду	100% покриття всієї бази коду
Виявлення ерозії	Важко помітити дрібні накопичення	Виявляє найменші відхилення від графа

Завдяки інтеграції цього алгоритму в процес розробки, команди можуть підтримувати «чистоту» архітектури протягом усього життєвого циклу проекту, запобігаючи перетворенню системи на «велику купу бруду» (Big Ball of Mud).

3.4. Порівняльний аналіз технік моделювання рефлексії та правил залежностей

Моделі рефлексії (Reflection Models) є універсальною технікою SACC, не обмеженою специфікою шаблону MVC. Важливою перевагою даного підходу є візуальне моделювання, що дозволяє інтуїтивно зіставляти реалізовану структуру з наміченою архітектурою (рис. 2.8). Однак, візуальна репрезентація стає деструктивним фактором при описі складних зовнішніх інтерфейсів (UIAPI): за відсутності суворої структуризації API граф стає перенасиченим («візуальний безлад»). Порівняно з правилами залежностей, моделі рефлексії мають нижчу експресивність, що ускладнює аналіз гетерогенних середовищ. Крім того, техніка схильна до пропуску порушень (false negatives): достатньо імплементації лише однієї легітимної залежності для архітектурного елемента «View», щоб уся сукупність релевантних йому класів вважалася відповідною специфікації. Дана проблема обмеженої

чутливості до виявлення компонентів для повторного використання підтверджується комерційними кейс-стаді.

Правила залежностей (Dependency Rules) також належать до загальних методів SACC. На сьогодні не зафіксовано високорівневих реалізацій правил, специфікованих виключно під MVC; найбільш абстрактною одиницею в інструментах на кшталт HUSACCT залишається концепт «шару» (layer). Традиційно правила формулюються загальними мовами запитів, що забезпечує високу гнучкість при роботі з еволюціонуючими UIAPI. Правила ітеруються для кожної одиниці коду, що гарантує високу точність виявлення кандидатів на рефакторинг у межах шару «View». Проте для повної верифікації системи з N вузлів потрібно $N \times (N-1)$ предикатів, що створює значні організаційні витрати при масштабуванні архітектури.

Моделі рефлексії забезпечують кращу абстракцію та інтуїтивність, тоді як правила залежностей пропонують вищу експресивність та легкість інтеграції в автоматизовані сервіси через їхній текстовий формат. Основним недоліком моделей рефлексії залишається ризик хибнонегативних результатів, а правил — низький рівень абстракції. Оптимальним рішенням є розробка методу, що базується на правилах, але впроваджує вищі рівні архітектурної семантики.

3.4.1 Метод SACC на основі архітектурних шаблонів

Для задоволення вимог, ідентифікованих у розділі 2, розроблено метод архітектурної верифікації, що базується на правилах, але оперує концептами шаблонів. Пріоритетом методу є мінімізація накладних витрат на підтримку та простота адаптації для програмних сервісів.

В основі алгоритму (лістинг 3.2) лежать дві функції класифікації елемента вихідного коду (codeElem):

- DevC: класифікує елемент з позиції розробника (на основі іменування та просторів назв).

- ArchC: класифікує елемент на основі фактичних графів залежностей (позиція архітектора).

У разі невідповідності результатів (dc ≠ ac) функція DivAbs ідентифікує характер порушення (дивергенція або відсутність зв'язку).

Лістинг 3.2. Псевдокод алгоритму

```
forall codeElem in Project do
    /* Класифікація елемента вихідного коду з точки зору розробника */
    dc = DevC(codeElem);

    /* Класифікація елемента вихідного коду на основі залежностей коду */
    ac = ArchC(codeElem);

    if dc != ac then
        /* Знаходження дивергентних (відхилених) або відсутніх залежностей */
        DivAbs(dc, ac, codeElem);
    end
end
```

Для шаблону MVC ключовим є включення четвертого елемента — UIAPI (зовнішній інтерфейс), який слугує джерелом «істини» для функції ArchC. Класифікація здійснюється через метрику відстані (d) до UIAPI в графі залежностей:

d =1: View (пряме використання API).

d > 1: Controller (опосередковане використання).

d = ∞: Model (відсутність залежностей від UIAPI).

Цей евристичний підхід дозволяє уникнути потреби у створенні громіздких наборів правил (наприклад, шести обов'язкових правил для MVC), забезпечуючи при цьому високу діагностичну здатність.

Лістинг 3.3. Реалізація алгоритму на Python

```python
def check_architectural_conformance(project_elements):
    """
    Метод для архітектурної перевірки відповідності на основі шаблонів.
    """
    for code_elem in project_elements:
        # 1. Класифікація з точки зору розробника (через іменування/пакети)
        # Наприклад: елемент у пакеті 'view' розробник вважає частиною View
        dc = dev_c(code_elem)
```

```
        # 2. Класифікація з точки зору архітектора (через відстань до UIAPI)
        # На основі аналізу графа залежностей
        ac = arch_c(code_elem)

        # 3. Порівняння двох класифікацій
        if dc != ac:
            # Виявлення дивергентних або відсутніх залежностей
            div_abs(dc, ac, code_elem)

def arch_c(element):
    """
    Евристична класифікація на основі відстані до UIAPI:
    - відстань = 1 -> View
    - відстань > 1 -> Controller
    - немає зв'язку -> Model
    """
    distance = get_distance_to_uiapi(element)
    if distance == 1:
        return "View"
    elif distance > 1:
        return "Controller"
    else:
        return "Model"
```

Наведемо короткий опис логіки алгоритму.

Функція DevC: Класифікує елемент на основі того, як його бачить розробник (зазвичай за іменем або розташуванням у файловій системі).

Функція ArchC: Використовує фактичні залежності елемента для класифікації (позиція архітектора). У вашому випадку вона базується на відстані до UIAPI у графі залежностей.

Функція DivAbs: Визначає характер порушення, якщо погляди розробника та архітектора на роль елемента не збігаються.

3.4.2 Верифікація через гіпотетичні сценарії

Для оцінки методу використано додаток «Dice Game». Контекст UIAPI визначено командами консольного вводу/виводу.

Еталонна реалізація. Усі класи класифікуються ідентично обома функціями; порушень не виявлено.

Рисунок 3.2 ілюструє UML-діаграму класів архітектурно відповідної версії програмного продукту.

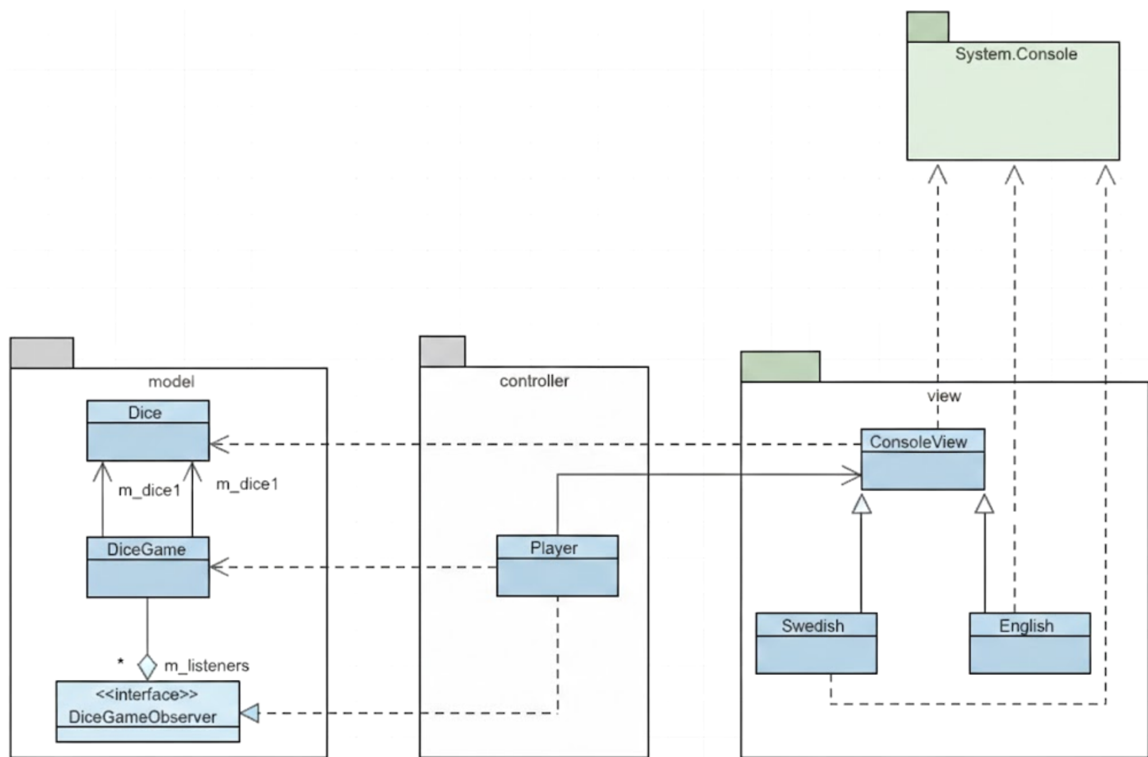

Рис. 3.2. UML-діаграма класів архітектурно відповідної версії

Дана діаграма зіставляється з графом залежностей сутностей вихідного коду (класів та інтерфейсів) та їхніх взаємозв’язків (асоціація, залежність, успадкування, реалізація). Компонент Модель (Model) включає три елементи вихідного коду, що забезпечують агрегацію двох об’єктів гральних костей, обчислення ігрових правил та передачу подій слухачам (listeners) у момент кидка кості відповідно до шаблону Observer. Компонент Вигляд (View) містить один абстрактний клас ConsoleView із двома конкретними імплементаціями — English та Swedish. Усі зазначені елементи використовують пакет System.Console для забезпечення операцій консольного введення (читання клавіш) та виведення (запис рядка).

Нарешті, компонент Контролер (Controller) представлений класом Player, який параметризується конкретним класом Вигляду, а також класом DiceGame, що реалізує інтерфейс DiceGameObserver для маршрутизації подій до рівня представлення під час ігрового процесу. Контролер забезпечує циклічне виконання ігрової логіки до моменту ініціації завершення сесії

користувачем. Усі типи об’єктів класифікуються як відповідні цільовим архітектурним елементам, внаслідок чого девіацій (порушень) не виявлено.

Дефект патерна Observer. Пряма агрегація Контролера в Моделі призводить до того, що клас Моделі отримує непряму залежність від UIAPI та класифікується як Контролер.

Припустимо, що розробник вносить оперативні виправлення, дозволяючи класу DiceGame безпосередньо містити класи контролера Player замість створення інтерфейсу Observer (рисунок 3.3).

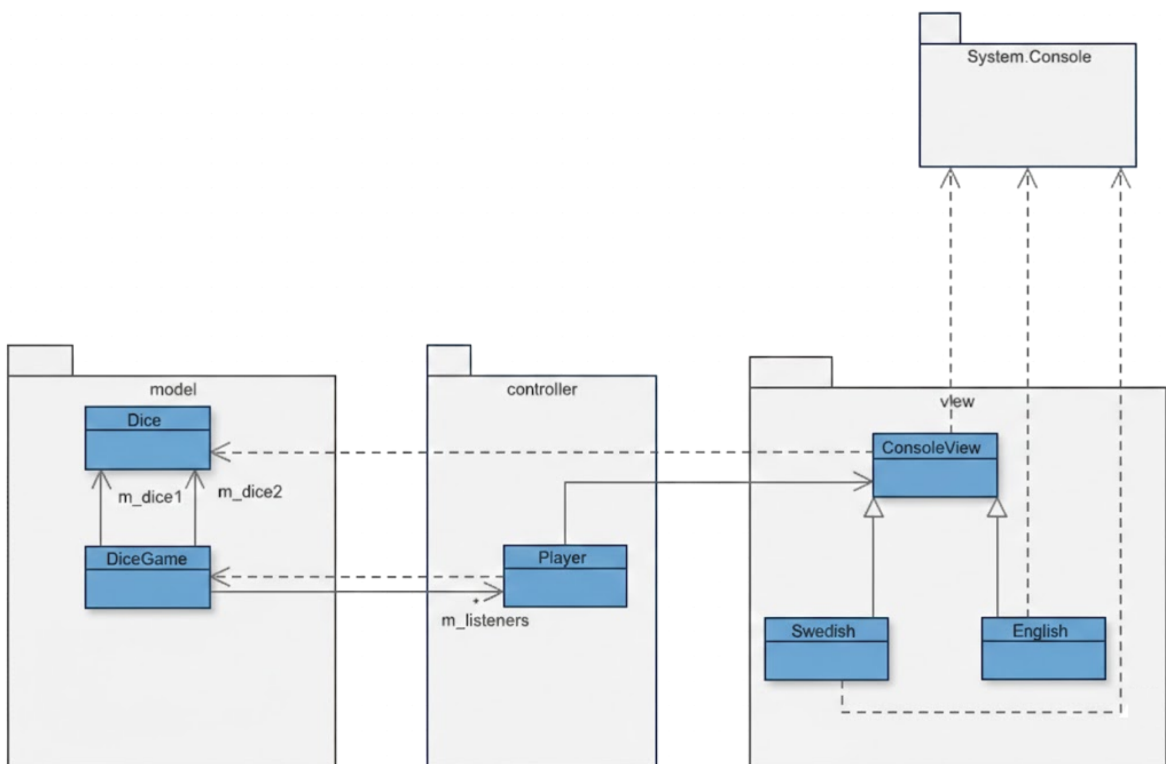

Рис. 3.3. UML-діаграма класів із помилковою реалізацією шаблону Observer

Дана маніпуляція формує непряму дивергентну залежність від DiceGame до UIAPI. У результаті запланований клас Моделі DiceGame буде класифікований системою як Контролер. Цей сценарій ілюструє механізм, що дозволяє уникнути виникнення циклічної залежності між архітектурними елементами Моделі та Контролера.

Порушення локації інтерфейсу. Розміщення інтерфейсу спостерігача в пакеті Контролера створює конфлікт між DevC та ArchC.

Дифузія відповідальностей. Використання викликів консолі в класі Player (Controller) або Dice (Model) призводить до їх помилкової класифікації як елементів View.

Ідентифікація компонентів для повторного використання: Клас у шарі View, який не залежить від конкретного UIAPI, класифікується як «Model», що вказує на його високий потенціал для абстрагування у вищий шар.

Припустимо, що в межах компонента View (Вигляд) створюється новий клас CoolConsole, який не має прямої залежності від System.Console, проте забезпечує узагальнену функціональність консольного інтерфейсу (рисунок 3.4).

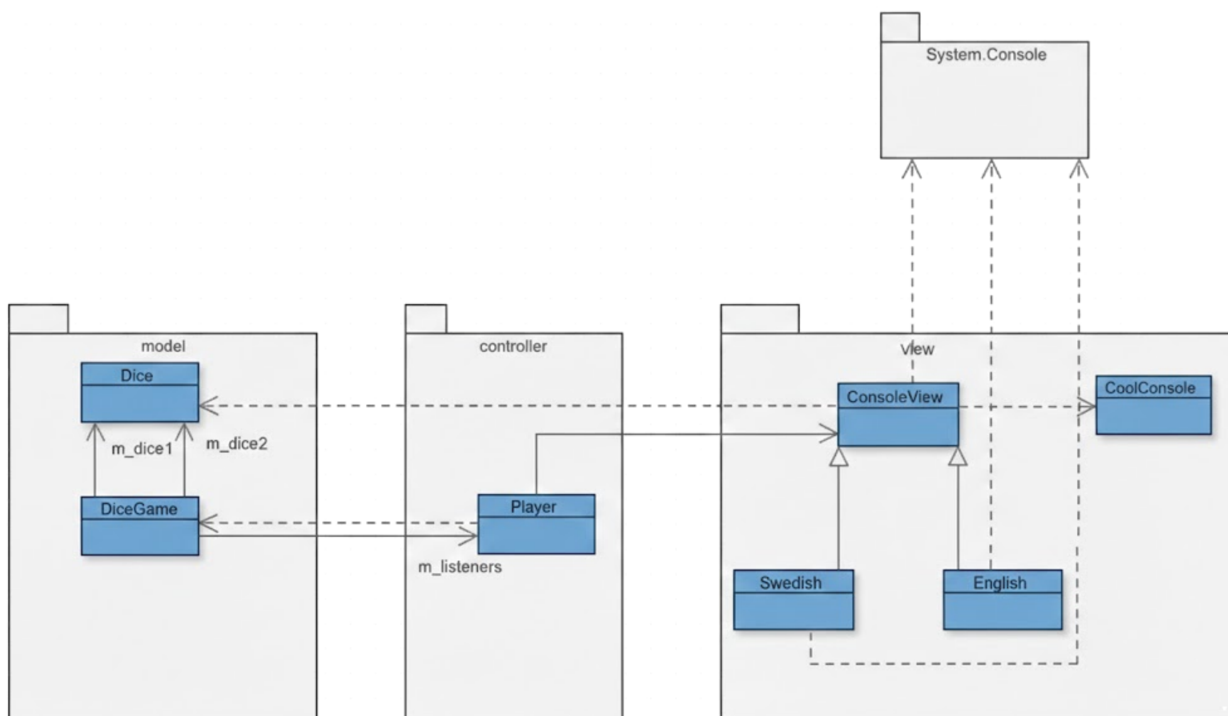

Рис. 3.4. UML-діаграма класів для сценарію виявлення компонента повторного використання у шарі View

Таким чином, даний клас розглядається як потенційний кандидат для повторного використання в контексті System.Console. Відповідно до алгоритму, цей клас буде класифікований як елемент Моделі (за критерієм

ArchC) та елемент Вигляду (за критерієм DevC) відповідно, що буде зафіксовано як архітектурне порушення.

Припустимо, що архітектор приймає рішення щодо впровадження нового архітектурного елемента. Даний елемент представлений класом-обгорткою ViewWrapper, відповідальним за інстанціювання конкретних об'єктів ConsoleView та надання уніфікованого інтерфейсу для взаємодії з елементами рівня Controller (Контролер) (рисунок 3.5).

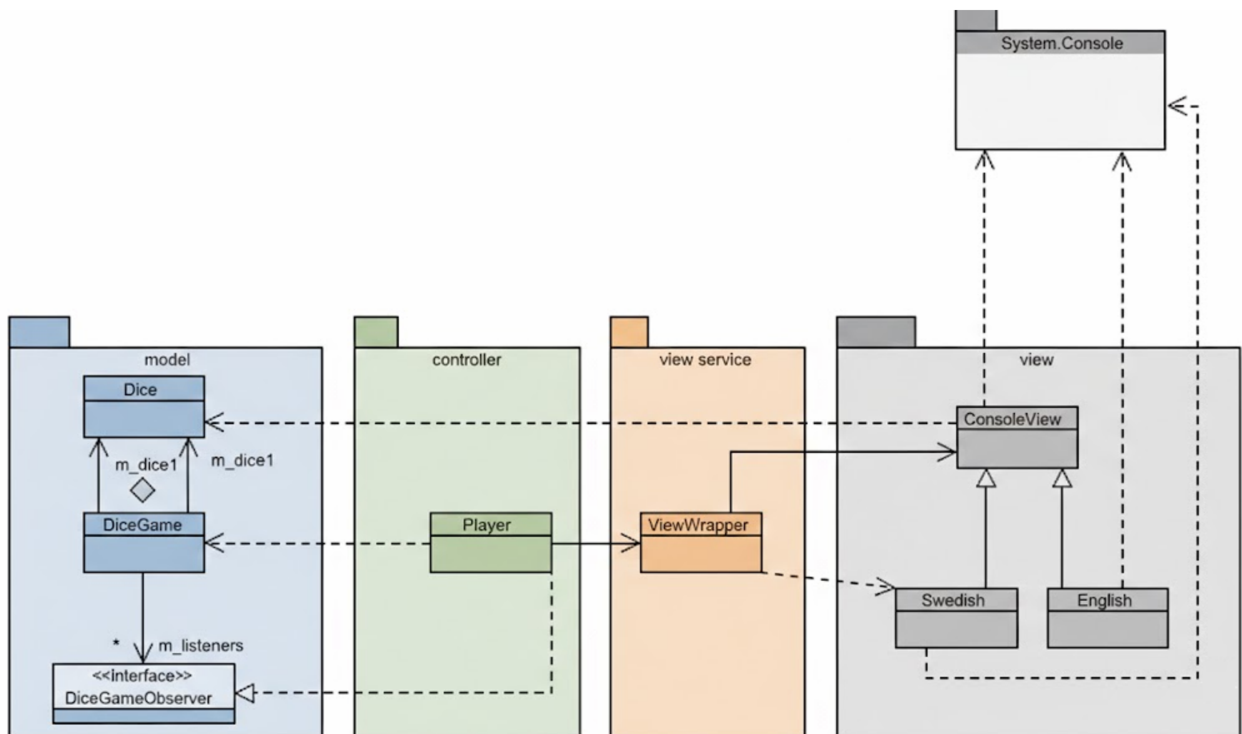

Рис. 3.5. UML-діаграма класів для сценарію впровадження нового архітектурного елемента

Реалізація цього рішення потребує модифікації метрики відстані в алгоритмі класифікації: визначення Контролерів за дистанцією d>2, елементів сервісу представлення (ViewService) — за d=2, а елементів Вигляду — за d=1.

Отже, розроблений метод SACC забезпечує ефективну верифікацію відповідності шаблону MVC шляхом дуальної класифікації елементів коду. Використання метрики відстані до фундаментальних архітектурних точок

(UIAPI) дозволяє автоматизувати виявлення структурної ерозії без необхідності ручного опису всіх можливих комбінацій залежностей. Метод вимагає мінімальної параметризації (визначення базисного API та порогів відстані), що робить його придатним для інтеграції в безперервні процеси архітектурного нагляду.

Запропонований метод SACC базується на евристичному підході, що передбачає ймовірність отримання неідеальних результатів. Існують граничні сценарії, за яких метод може демонструвати обмежену ефективність, ініціюючи пропуски фактичної архітектурної ерозії або генеруючи хибні повідомлення про порушення. У межах даного дослідження поставлено такі дослідницькі питання: оцінка загальної ефективності методу (RQ 1), ідентифікація необхідних рефакторингів для забезпечення відповідності кодової бази (RQ 2.1) та аналіз впливу цих рефакторингів на метрики розміру й зв'язності ПЗ (RQ 2.2). Крім того, аналізу підлягають типи ерозії, не виявлені методом (RQ 2.4), та можливості параметризації алгоритму для мінімізації хибнопозитивних результатів (RQ 2.3).

3.5. Методологія оцінки ефективності методу

Для верифікації ефективності методу та визначення стратегій рефакторингу проводиться компаративний аналіз результатів експертного ручного огляду та автоматизованого аналізу декількох проектів. У дослідженні використовуються такі дефініції:

- Ерозія: елемент вихідного коду з ознаками архітектурної деградації, що перебуває у стані переходу до нелегітимної залежності.

- Порушення: елемент, ідентифікований методом як такий, що піддався ерозії.

- TP - фактична ерозія, класифікована як порушення.

- FP - відсутність ерозії при наявності звіту про порушення.

FN - наявна ерозія, не ідентифікована як порушення.

Таблиця 3.2.

Протокол архітектурної інспекції

Тип ерозії	**Опис перевірки**
Нелегітимна залежність	Наявність зв'язків, що суперечать інваріантам шаблону MVC.
Логіка моделі в UI	Реалізація бізнес-правил або доменної логіки в шарі інтерфейсу.
Специфічна для UI модель	Адаптація компонентів моделі під потреби конкретного представлення (View).
Незв'язна модель	Наявність у моделі декількох слабопов'язаних зон відповідальності.
Замаскований тип	Приховування складних залежностей через узагальнені типи або ідентифікатори.

Для встановлення «основної істини» (ground truth) проводиться ручна інспекція кожного елемента вихідного коду. Процес регламентується протоколом (табл. 3.2), розробленим на основі аналізу типових архітектурних помилок у MVC-системах та емпіричного досвіду авторів.

Технічна реалізація методу здійснюється у напівавтоматичному режимі. За допомогою спеціалізованої мови запитів (Query Language) інструменту аналізу коду обчислюється метрика відстані від кожного елемента до UIAPI. Ці дані формують підґрунтя для архітектурної класифікації (ArchC). Класифікація розробника (DevC) виконується шляхом аналізу конвенцій іменування, просторів назв та функціонального призначення елементів (утримання стану, рендеринг або ігрові правила).

На основі отриманих статистичних даних (TP, FP, FN) обчислюються ключові метрики:

- Точність (Precision, P): відношення підтверджених випадків ерозії до загальної кількості виявлених порушень. Низький показник P свідчить про надмірну кількість хибних тривог, що знижує продуктивність розробників.

$$P = \frac{TP}{TP + FP}$$

Повнота (Recall, R): частка виявленої ерозії серед усіх фактично наявних дефектів. Низький показник R вказує на низьку чутливість методу до архітектурних відхилень.

$$R = \frac{TP}{TP + FN}$$

Аналізу підлягають усі виявлені порушення (TP та FP), для яких пропонуються реалістичні сценарії рефакторингу або коригування параметрів методу. Після імплементації змін проводиться повторний цикл SACC-аналізу. Верифікація включає порівняння архітектурно невідповідної та скоригованої версій ПЗ.

Для оцінки впливу архітектурних змін на якість коду використовуються метрики вихідних залежностей (Efferent coupling / Fan-out) та обсягу коду (Lines of Code, LoC), що є релевантними індикаторами підтримуваності системи. Оцінка результатів рефакторингу базується на методології Буркена та Келлера, де архітектурна ерозія виступає основним драйвером технічних змін. Порівняння станів проекту «до» та «після» здійснюється з використанням систем контролю версій (VCS).

3.6. Експериментальна оцінка методу

Для проведення експериментального аналізу було відібрано чотири програмні продукти, що реалізують архітектурний шаблон MVC. Вибір вибірки здійснювався з урахуванням критеріїв життєвого циклу, жанрової належності та ступеня комерційної зрілості систем. Усі об'єкти дослідження розроблені на мові C++ з використанням об'єктно-орієнтованої парадигми в середовищі Visual Studio. Системи базуються на спільному ігровому рушії (EngineIII), який забезпечує низькорівневу функціональність та рендеринг через API DirectX.

Статистичні характеристики проектів, включаючи обсяг коду (LoC), виміряний за допомогою утиліти CLOC, та кількість типів, визначену інструментом CPPDepend, наведено в таблиці 3.3.

Таблиця 3.3.

Метрики розміру та жанрова класифікація проектів

Жанр проекту	**LoC**	**Кількість типів**
Ігровий рушій	46,390	225
3D платформа	5,362	56
Головоломка	12,046	68
Ізометрична RPG	21,639	167
Shoot 'Em Up	14,212	137

На рисунку 3.6 представлена цільова архітектура систем із визначеним зовнішнім інтерфейсом UIAPI (dxrnd). Для зниження когнітивної складності аналізу деякі специфічні доменні елементи рушія були виключені з архітектурної моделі.

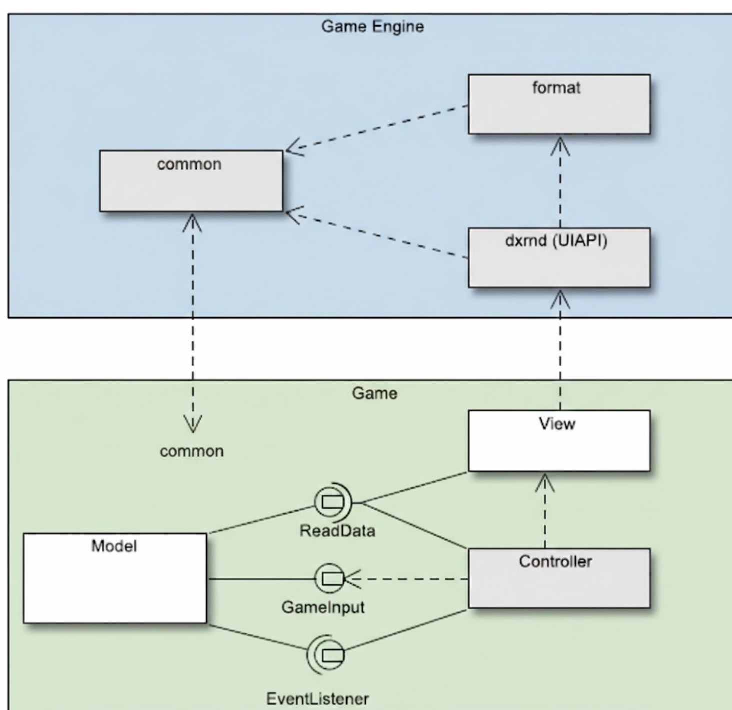

Рис. 3.6. Цільова архітектура систем із визначеним зовнішнім інтерфейсом

3.6.1 Процедура класифікації та верифікації

Для встановлення «основної істини» (ground truth) було проведено ручний огляд кожної сутності. Класифікація розробника (DevC) базувалася на аналізі семантики іменування, просторів назв та функціональних обов'язків (state management, rendering, game rules).

Паралельно було виконано автоматизовану класифікацію (ArchC) за допомогою мови запитів LINQ в середовищі CPPDepend (лістинг 3.4). Алгоритм обчислював глибину залежностей кожного типу від простору назв dxrnd. Елементи з дистанцією d=1 ідентифікувалися як View, d>1 — як Controller, а автономні елементи (d=0) — як Model.

Лістинг 3.4. Запит на мові LINQ для детермінації архітектурної дистанції до UIAPI

```
from t in Application.Types
where t.ParentProject.Name == "ProjectName"
let d = t.DepthOfIsUsing("dxrnd")
select new { t, d }
```

Порівняння результатів DevC та ArchC дозволило ідентифікувати архітектурні порушення. Статистичні показники ефективності методу (точність та повнота) наведені в таблиці 3.5.

Таблиця 3.5.

Показники ефективності результатів класифікації

Жанр проекту	**TP**	**FP**	**FN**	**Точність (P)**	**Повнота (R)**
Ігровий рушій	10	1	6	0.90	0.63
3D платформа	15	3	2	0.83	0.88
Головоломка	28	9	8	0.76	0.79
Ізометрична RPG	23	5	9	0.82	0.72
Всього	**76**	**18**	**25**	**0.81**	**0.75**

3.6.2 Рефакторинг та усунення ерозії

Для апробації стратегій рефакторингу було обрано два проекти: 3D platform (активна фаза розробки) та Shoot ’Em Up (завершений проект із високим ступенем ерозії). Аналіз виявив відсутність порушень у шарі Моделі; девіації локалізувалися переважно у взаємодії між Виглядом та Контролером. Типологію запропонованих заходів наведено в таблиці 3.6.

Таблиця 3.6.

Розподіл стратегій рефакторингу в проектах

Тип рефакторингу	3D platform	Shoot ’Em Up	Опис дії
Розділення типу	0	3	Декомпозиція еродованого класу на View та Controller
Переміщення типу	6	6	Міграція сутності до відповідного компонента
Переміщення функції	0	4	Релокація методу зі змішаною відповідальністю
Інкапсуляція типу	1	10	Агрегація допоміжних типів у батьківські структури
Уточнення методу	1	5	Коригування параметрів запиту SACC

Рефакторинг проекту Shoot ’Em Up був зосереджений на розділенні еродованих класів. Зокрема, ідентифіковані кандидати на повторне використання (обробка DirectInput) були інкапсульовані в межах простору назв dxrnd.

Аналіз помилок класифікації:

- Хибні позитивні (FP): У проекті Shoot ’Em Up викликані використанням типів даних поза межами UIAPI. У 3D platform зафіксовано одиничний збій інструментарію при детекції залежностей.

- Хибні негативні (FN): Більшість випадків (18 з 25) зумовлені обмеженим визначенням UIAPI. Специфічні проблеми, такі як реалізація ігрових правил безпосередньо в UI (Puzzle) або передача аудіо-

ідентифікаторів від Моделі до UI (Isometric RGP), залишаються складними для автоматизованого виявлення.

3.6.3 Вплив на структурні метрики

Після проведення рефакторингу в проекті Shoot ’Em Up було зафіксовано зміну показників зв'язності (Coupling). Результати порівняльного аналізу до та після втручання представлені в таблиці 3.7.

Таблиця 3.7.

Динаміка зв'язності (Efferent Coupling) у Shoot ’Em Up проекті

Клас	До рефакторингу	Після рефакторингу
GameView	64	46
ViewCore	45	53
ViewManager	28	16
GameController	—	37
Середнє значення	**19.87**	**18.14**

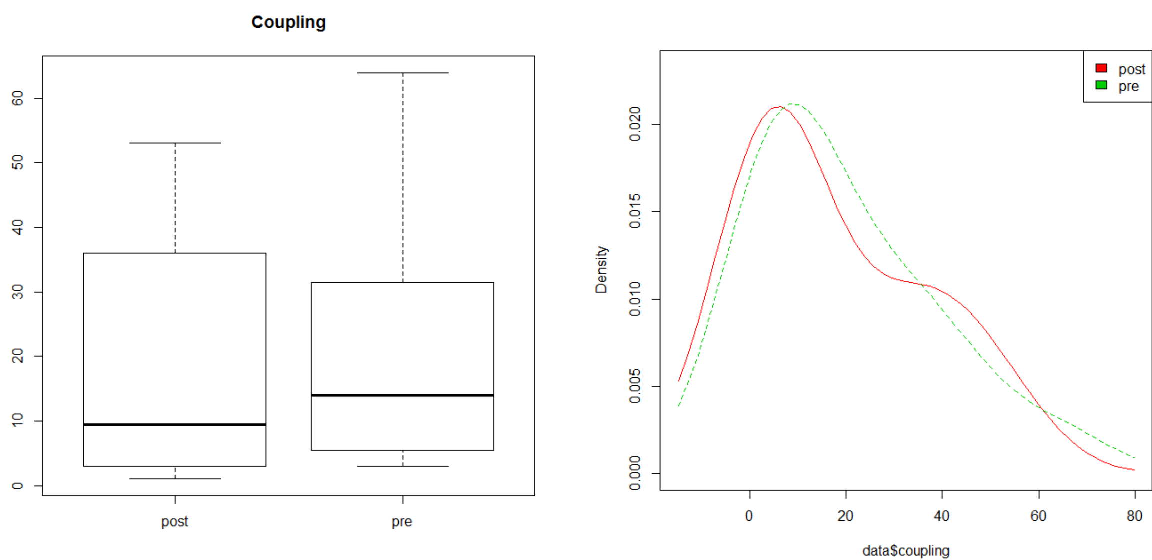

Рис. 3.7. Динаміка змін показників зв'язності (Coupling) у Shoot ’Em Up проекті до та після рефакторингу

Аналіз щільності розподілу зв'язності (рисунок 3.7) демонструє зменшення екстремальних значень та більш рівномірний розподіл залежностей. Зокрема, зв'язність найбільш складного класу GameView знизилася на 28%. Дані щодо обсягу коду (LoC) підтверджують, що рефакторинг призвів до перерозподілу логіки: обсяг шару Контролера зріс з 287 до 664 операторів за рахунок розвантаження шару Вигляду.

Отже, експериментально підтверджено ефективність методу SACC з інтегральним показником точності 0.81 та повноти 0.75. Впровадження архітектурно зумовленого рефакторингу сприяє покращенню підтримуваності системи шляхом децентралізації зв'язності та чіткого розподілу відповідальностей між елементами MVC. Для підвищення точності методу рекомендується динамічне розширення визначення UIAPI та впровадження фільтрації типів за пороговим значенням LoC.

Ефективність методу була перевірена на п’яти ігрових проектах різного ступеня зрілості (від 5,000 до 46,000 LoC). Основні результати аналізу:

- Діагностична здатність: Середній показник точності (Precision) склав 0.81, а повноти (Recall) — 0.75. Це підтверджує здатність методу виявляти більшість критичних архітектурних аномалій при низькому рівні хибних тривог.

- Ідентифікація ерозії: Найпоширенішими типами девіацій виявилися змішування відповідальностей між View та Controller та прямі залежності Моделі від UIAPI через "замасковані типи".

- Виявлення потенціалу повторного використання: Метод успішно ідентифікував класи у шарі View, які були кандидати на міграцію до загального ігрового рушія.

Проведено архітектурно обумовлений рефакторинг еродованих систем, результати якого продемонстрували значне покращення структурних метрик:

- Зниження зв’язності: У найбільш критичних класах (наприклад, GameView) показник Efferent Coupling знизився на 28%.

- Перерозподіл відповідальності: Обсяг коду (LoC) у шарі Контролера зріс у 2.3 рази за рахунок винесення логіки з шару Представлення, що привело систему до цільового стану шаблону MVC.

- Стабілізація структури: Діаграми щільності підтвердили більш рівномірний розподіл залежностей, що безпосередньо корелює зі зниженням технічного боргу та покращенням підтримуваності.

Впровадження запропонованого методу дозволить перетворити архітектурний контроль з епізодичного процесу на безперервну перевірку, забезпечуючи життєздатність складних програмних систем протягом тривалого циклу розробки.

Висновки до розділу

У третьому розділі реалізовано формалізований метод забезпечення архітектурної цілісності програмних систем. Розроблено алгоритмічну основу для виявлення порушень архітектурних інваріантів. Запропоновано класифікацію типових форм архітектурної ерозії в межах шаблону MVC. Побудовано математичну модель перевірки залежностей між компонентами системи. Деталізовано метод верифікації архітектури на основі графової декомпозиції залежностей. Здійснено порівняльний аналіз технік моделювання рефлексії та формалізованих правил залежностей. Показано переваги методу SACC з точки зору точності та інтерпретованості результатів. Розроблено методологію оцінки ефективності запропонованого підходу. Проведено експериментальну перевірку методу з використанням процедур класифікації та рефакторингу. Отримані результати підтвердили позитивний вплив формалізованої верифікації на структурні характеристики програмних систем.

ВИСНОВКИ

У магістерській роботі здійснено дослідження формальних методів та інструментальних засобів забезпечення архітектурної цілісності програмних систем в умовах їх еволюції та зростаючої складності. Основну увагу зосереджено на проблемі архітектурної ерозії, що виникає внаслідок розбіжності між задекларованою архітектурною моделлю та її фактичною реалізацією у програмному коді.

У першому розділі виконано теоретичне обґрунтування проблеми архітектурної цілісності програмних систем та проаналізовано сучасні підходи до верифікації архітектурних шаблонів. Встановлено, що традиційні емпіричні та інтуїтивні методи контролю архітектури є недостатніми для складних програмних систем, які зазнають постійної еволюції.

Показано, що архітектурні шаблони доцільно розглядати як когнітивні одиниці, які фіксують інваріанти структури та поведінки системи. Це дозволяє формалізувати очікувану архітектурну відповідність та застосовувати методи автоматизованої перевірки. Розглянута методологія Architectural Compliance Checking (ACC) довела свою ефективність як концептуальна основа для виявлення порушень архітектурних обмежень.

Особливу увагу приділено гібридному підходу до архітектурної верифікації, який поєднує формальні методи, експертні правила та аналітичні моделі. Запропоновано математичну формалізацію архітектурної відповідності з використанням матричного апарату, що дозволяє виявляти структурні аномалії та класифікувати типові порушення залежностей. Отримані результати створюють теоретичне підґрунтя для подальшої реалізації формалізованих методів контролю архітектурної цілісності.

Другий розділ присвячено дослідженню формальних методів та інструментів аналізу емпіричних проявів архітектурної ерозії на прикладі шаблону Model–View–Controller. Проведено контекстуалізацію MVC як

архітектурного шаблону та проаналізовано його трансформацію в процесі розвитку програмних систем, зокрема у сфері ігрової розробки.

Ретроспективний аналіз еволюції архітектури продемонстрував, що відсутність формалізованих механізмів контролю призводить до накопичення технічного боргу та поступової деградації архітектурної структури. Запропонований процес семантичного анотування компонентів системи дозволяє встановити явний зв’язок між елементами коду та їх архітектурними ролями, що є критично важливим для автоматизованої верифікації.

Побудова графіків еволюції технічного боргу підтвердила залежність між жанровими особливостями проєктів та інтенсивністю архітектурної ерозії. Розглянута методологія статичної перевірки архітектурної відповідності на основі SACC (Static Architectural Compliance Checking) показала високу придатність для формалізованого контролю залежностей. Обґрунтовано роль архітектурних інспекцій та формалізації правил залежностей як ключових інструментів забезпечення цілісності системи на етапах її розвитку.

У третьому розділі реалізовано формалізований метод верифікації архітектурної цілісності та виконано його експериментальну оцінку. Запропоновано алгоритмічну базу методу, що дозволяє виявляти порушення архітектурних інваріантів на основі аналізу залежностей між компонентами системи.

Розроблено математичну модель перевірки залежностей, яка ґрунтується на графовій декомпозиції та формалізованих правилах взаємодії. Деталізовано метод верифікації архітектурних інваріантів, який забезпечує виявлення типових проявів ерозії в архітектурі MVC. Порівняльний аналіз технік моделювання рефлексії та правил залежностей показав переваги підходу SACC з точки зору масштабованості та інтерпретованості результатів.

Експериментальна оцінка підтвердила ефективність запропонованого методу щодо класифікації порушень, підтримки процесів рефакторингу та зменшення рівня архітектурної ерозії. Встановлено позитивний вплив застосування формалізованої верифікації на структурні метрики програмних систем, зокрема на зменшення циклічних залежностей та покращення модульності.

У результаті виконання магістерської роботи досягнуто поставленої мети — розроблено та обґрунтовано формалізований підхід до забезпечення архітектурної цілісності програмних систем на основі поєднання архітектурних шаблонів, математичних моделей та методів статичної верифікації. Отримані результати мають як теоретичну, так і практичну цінність та можуть бути використані для підвищення якості архітектурних рішень у сучасних програмних проєктах.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Kruchten, P. (1995). The 4+1 view model of architecture. IEEE Software, 12(6), 42–50. https://doi.org/10.1109/52.469759
2. Bass, L., Clements, P., & Kazman, R. (2003). Software architecture in practice. Addison-Wesley.
3. Shaw, M., & Garlan, D. (1996). Software architecture: Perspectives on an emerging discipline. Prentice Hall.
4. Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 26(1), 70–93. https://doi.org/10.1109/32.825767
5. Kazman, R., Klein, M., & Clements, P. (2000). ATAM: Method for architecture evaluation. Carnegie Mellon University, SEI Technical Report.
6. Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. IEEE Software, 29(6), 18–21. https://doi.org/10.1109/MS.2012.167
7. Avgeriou, P., Kruchten, P., Ozkaya, I., & Seaman, C. (2016). Managing technical debt in software engineering. IEEE Software, 33(2), 87–90. https://doi.org/10.1109/MS.2016.50
8. Martini, A., Bosch, J., & Chaudron, M. R. V. (2015). Architectural technical debt: Understanding causes and a qualitative model. Proceedings of the 37th ICSE. https://doi.org/10.1109/ICSE.2015.35
9. Li, R., Liang, P., & Avgeriou, P. (2021). Architectural technical debt identification: A systematic mapping study. Journal of Systems and Software, 173, 110885. https://doi.org/10.1016/j.jss.2020.110885
10. Suryanarayana, G., Samarthyam, G., & Sharma, T. (2014). Refactoring for software design smells. Morgan Kaufmann.
11. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern-oriented software architecture: A system of patterns. Wiley.

12. Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California, Irvine.
13. Mens, T., & Tourwé, T. (2004). A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2), 126–139. https://doi.org/10.1109/TSE.2004.1265817
14. Babar, M. A., & Gorton, I. (2009). Comparison of scenario-based software architecture evaluation methods. Proceedings of the 11th ECSA. https://doi.org/10.1007/978-3-642-10447-6_11
15. Nord, R. L., & Tomayko, J. E. (2004). Software architecture: A roadmap. Proceedings of the Future of Software Engineering. https://doi.org/10.1145/2593882.2593884
16. Razavian, M., & Lago, P. (2011). A frame of reference for SOA governance. Proceedings of the WICSA. https://doi.org/10.1109/WICSA.2011.21
17. O'Neill, P., & Laplante, P. A. (2019). Architectural conformance checking: A systematic review. Journal of Systems and Software, 151, 1–17. https://doi.org/10.1016/j.jss.2019.01.030
18. Murphy, G. C., Notkin, D., & Sullivan, K. (2001). Software reflexion models. ACM Transactions on Software Engineering and Methodology, 10(4), 359–409. https://doi.org/10.1145/504183.504185
19. Knodel, J., & Popescu, D. (2007). A comparison of static architecture compliance checking approaches. Proceedings of WCRE. https://doi.org/10.1109/WCRE.2007.29
20. Brunet, J., & Medvidovic, N. (2014). Software architecture reconstruction. IEEE Software, 31(4), 80–87. https://doi.org/10.1109/MS.2014.80
21. Zazworka, N., Shaw, M., Shull, F., & Seaman, C. (2013). Investigating the impact of design debt on software quality. Proceedings of ICSM. https://doi.org/10.1109/ICSM.2013.37

22. Digkas, G., Lungu, M., Avgeriou, P., & Spinellis, D. (2018). How do developers fix architectural smells? Proceedings of SANER. https://doi.org/10.1109/SANER.2018.8330221
23. Lenarduzzi, V., Arcelli Fontana, F., & Taibi, D. (2018). Does architectural smell severity matter? Proceedings of ECSA. https://doi.org/10.1007/978-3-030-00761-4_7
24. N. Sangal, E. Jordan, J. Sinha, and D. Jackson, "Using Dependency Models to Manage Software Architecture," in Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005.
25. Martini, A., Bosch, J., & Chaudron, M. (2017). Technical debt control. Journal of Systems and Software, 123, 45–61. https://doi.org/10.1016/j.jss.2016.10.019
26. Lehman, M. M., & Ramil, J. F. (2001). Software evolution in the age of component-based development. Proceedings of IWPSE. https://doi.org/10.1145/602461.602463
27. D. C. Pruijt, C. Koppe, and S. Brinkkemper, "Architecture Compliance Checking with Dependency Rules," in Proceedings of the 4th International Workshop on Software Quality and Maintainability, 2010.
28. Baldassarre, M. T., & Bianchi, A. (2014). Architecture erosion metrics. Journal of Software: Evolution and Process, 26(7), 673–692. https://doi.org/10.1002/smr.1633
29. D. C. Pruijt, C. Koppe, and S. Brinkkemper, "Static Architecture Compliance Checking: An Exploratory Case Study with HUSACCT," Utrecht University, Technical Report, 2014.
30. Candela, I., Bavota, G., Russo, B., & Oliveto, R. (2016). Using cohesion metrics to predict architectural smells. Proceedings of ICSME. https://doi.org/10.1109/ICSME.2016.51

31. Garcia, J., Popescu, D., Edwards, G., & Medvidovic, N. (2009). Identifying architectural bad smells. Proceedings of CSMR. https://doi.org/10.1109/CSMR.2009.29
32. Kruchten, P. (2009). Architecture and technical debt. Proceedings of WICSA. https://doi.org/10.1109/WICSA.2009.5290670
33. Lenarduzzi, V., Saarimäki, N., & Taibi, D. (2020). Architecture debt: A systematic mapping study. Journal of Systems and Software, 169, 110706. https://doi.org/10.1016/j.jss.2020.110706
34. Roveda, R., Arcelli Fontana, F., & Zanoni, M. (2018). Towards an architectural smell classification. Proceedings of SEKE.
35. M. de Jonge and H. J. van der Linden, "Defining and Checking Software Architectures," in Proceedings of the Joint 10th European Software Engineering Conference (ESEC), 2005.
36. J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in Proceedings of the IEEE/IFIP Working Conference on Software Architecture (WCSA), 2007.
37. Zdun, U., Avgeriou, P., & Dustdar, S. (2013). Architectural decisions as reusable assets. IEEE Software, 30(1), 64–69. https://doi.org/10.1109/MS.2012.164
38. Guimarães, E., & Garcia, A. (2017). Architectural erosion in long-lived systems. Journal of Software: Evolution and Process, 29(6), e1878. https://doi.org/10.1002/smr.1878
39. Ouni, A., Kula, R. G., & Inoue, K. (2017). Improving software architecture quality through refactoring. Empirical Software Engineering, 22(5), 2369–2406. https://doi.org/10.1007/s10664-016-9485-4
40. G. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," IEEE Transactions on Software Engineering, vol. 27, no. 4, 2001.
41. Zimmermann, O. (2010). Architectural decision modeling. IEEE Software, 28(2), 74–80. https://doi.org/10.1109/MS.2010.123

42. L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rd ed., Addison-Wesley Professional, 2012.
43. L. Passos, R. Terra, R. Valente, et al., "JITACC: A tool for Just-in-time Architecture Conformance Checking," in Proceedings of the 2013 IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR), 2013.