ISSN 2519-2310 (Online)

КОМП'ЮТЕРНІ НАУКИ ТА КІБЕРБЕЗПЕКА, 2024, № 1 (25)

COMPUTER SCIENCE AND CYBERSECURITY, 2024, Iss. 1 (25)

DOI: https://doi.org/10.26565/2519-2310-2024-1-07
УДК 004.056.5

РОЗРОБКА ТА РЕАЛІЗАЦІЯ МЕТОДА ПЕРЕВІРКИ ЦІЛІСНОСТІ ДИЗАЙНУ ОБ'ЄКТНО-ОРІЄНТОВАНОЇ СИСТЕМИ

Микита Пугач[1], аспірант кафедри теоретичної та прикладної інформатики, факультет математики та інформатики, e-mail: nikita.pugach.2000@gmail.com,
ORCID: https://orcid.org/0009-0004-8923-6489
Ірина Зарецька[1], доцент кафедри теоретичної та прикладної інформатики, факультет математики та інформатики, e-mail: zaretskaya@karazin.ua,
ORCID: https://orcid.org/0000-0001-8747-2737
[1]*Харківський національний університет імені В.Н. Каразіна, майдан Свободи, 4, Харків, 61022, Україна*

Рукопис надійшов 3 квітня 2024 р. Отримано після рецензування 5 травня 2024 р.
Прийнято 7 червня 2024 р.

Анотація: Для досягнення якості створення програмних продуктів, необхідно проводити різні заходи із тестування та верифікації на всіх етапах розробки, що є невід'ємним та одним з найважливіших етапів проектування ПЗ. У більшості моделей життєвого циклу програмного забезпечення (SDLC) даний етап є одним із перших, тож помилки, допущені при розробці дизайну приведуть до проблем у всіх наступних стадіях. Таким чином, через велику ціну помилки, важливою є перевірка цілісності розробленого дизайну на етапі проектування. У статті досліджується проблема пошуку протиріч у об'єктно-орієнтованому дизайні. Автори презентують набір протиріч, що можуть виникати у такому дизайні і ставлять за мету розробку методів виявлення та пошуку цих протиріч з метою покращення якості проектування, а також написання програмного забезпечення, що буде реалізовувати дані методи. Інструментом створення об'єктно-орієнтованого дизайну було обрано програму «diagrams.net», головною корисною рисою якої є можливість представлення створених діаграм у виді XML файлу у популярному форматі drawio. Автори пропонують метод за яким проводиться парсинг XML файлу діаграми і представлення її у виді набору об'єктів, таких як стрілки залежностей, класи, методи і т.д. Ці об'єкти повинні взаємодіяти за встановленими правилами. Порушення даних правил і є протиріччям об'єктно-орієнтованого дизайну. У результаті дослідження було представлено метод пошуку протиріч і реалізовано його на мові програмування Java.

Ключові слова: *розробка програмного забезпечення, UML, проектування програмного забезпечення, діаграми об'єктно-орієнтованого дизайну*

Як цитувати: Пугач М., Зарецька І.. Розробка та реалізація метода перевірки цілісності дизайну об'єктно-орієнтованої системи. *Комп'ютерні науки та кібербезпека.* 2024; № 1(25): С. 76–87. https://doi.org/10.26565/2519-2310-2024-1-07

In cites: Pugach M., Zaretska I. (2024). Development and implementation of a method for checking the integrity of the design of an object-oriented system. *Computer Science and Cybersecurity.* 1(25): 76–87. https://doi.org/10.26565/2519-2310-2024-1-07 (in Ukrainian)

1. Вступ

Проектування будь-якого програмного продукту є одним з найважливіших етапів розробки. Основна його мета полягає у створенні моделі майбутньої системи, яка буде визначати основні складові та взаємозв'язки. Результати проектування слугують фундаментом для подальшого написання програмного коду. Оскільки цей етап є дуже важким та витратним процесом розробки, дуже важливо максимально його оптимізувати та знайти найефективніші рішення для задоволення всіх вимог системи. Виявлення помилок, допущених на етапі проектування можуть у подальшому призвести до таких серйозних проблем, як збільшення часу виконання проекту, вслід чого збільшення його вартості, суттєве зниження якості готового програмного продукту, погіршення продуктивності системи або поганої масштабованості. Вкрай важливо приділяти багато уваги тому, щоб етап проектування не мав помилок, намагатися їх мінімізувати.

Стандартом у проектуванні програмного забезпечення є використання UML (Unified Modeling Language або уніфікованої мови моделювання). Він використовується для візуального представлення та документування програмних систем. UML надає нотацію та набір графічних символів для моделювання різних аспектів системи, таких як структура, функціональність, поведінка та взаємодія між компонентами.

В об'єктно орієнтованому дизайні може бути два типи протиріч. Перший, це коли протиріччя існує у рамках однієї діаграми. Другий, це протиріччя, що виникає на ґрунті несумісності двох UML діаграм. Такого виду протиріччя, що виникають на перетині двох або більше діаграм, не відслідковуються специфікацією UML, а отже не можуть бути перевірені Case-засобами. Протиріччя всередині однієї діаграми можуть виникати через те, що специфікація UML велика і Case-засоби не перевіряють повністю весь список вказаних там правил. А також існують ситуації, які виникають в рамках однієї діаграми і не описані в специфікації UML, але є несумісними з об'єктно орієнтованим дизайном. Наприклад, якщо в діаграмі класів є цикл залежностей, то це свідчить про те що вона була погано спроектована. Такі протиріччя також не можуть бути перевірені Case-засобами.

Мета даної статті це розробка методів виявлення та пошуку протиріч в об'єктно орієнтованому дизайні з метою покращення якості проектування, а також написання програмного забезпечення, що буде реалізовувати дані методи.

У статті розглядається проектування за допомогою мови моделювання UML. Чотири типи діаграм, що будуються з її допомогою складають основу на етапі проектування, це діаграми: класів, послідовностей, об'єктів та станів. Дослідження стосуються пошуку протиріч та несумісності саме в цих чотирьох діаграмах та між ними.

2. Огляд існуючих рішень

Для створення дизайну об'єктно-орієнтованих систем використовуються Case-засоби. Деякі сучасні Case-засоби, наприклад Rational Rose [11], мають можливість перевірки низки порушень цілісності дизайну на основі специфікації UML. Але вони є обмеженими і не надають можливість перевіряти цілісність між двома діаграмами різних видів.

Дослідження можливості верифікації цілісності між декількома видами діаграм проводилось у статтях «Cross-Diagram UML Design Verification» [9] та «Consistency of UML Design» [10]. Ці дослідження представляли дизайн у виді графової моделі та використовували логіку предикатів 1 порядку для пошуку протиріч, а також реалізовували пошук цих протиріч на мовах програмування Prolog та Java. Дане дослідження використовує більш простий підхід до пошуку порушень цілісності ніж вищеперераховані.

3. Опис протиріч

3.1. Протиріччя всередині діаграми класів, або між діаграмами класів

Діаграма класів – це UML-діаграма, яка описує систему, візуалізуючи різні типи об'єктів усередині системи та види статичних зв'язків, що існують між ними. Він також ілюструє операції та атрибути класів. Тому вкрай важливо звести до мінімуму протиріччя в цій діаграмі. В результаті аналізу специфікації UML 2.0 було знайдено такі види можливих протиріч:

- Клас має два або більше стереотипи, що не є сумісними між собою. Наприклад, клас одночасно позначений стереотипами «exception» та «enumeration».
- У випадку композиції кожен об’єкт-частина може належати тільки одному об’єкту-цілому. Для того, щоб ця вимога дотримувалася у діаграмі повинні бути враховані такі вимоги:
    - Кратність кінця-цілого при композиції не повинна перевищувати одиниці;
    - Один клас не може бути частиною більше ніж однієї композиції.
- Клас, що має стереотип «utility», не може містити у собі методи або атрибути з областю видимості екземпляру [4, с.682].
- Клас успадковує клас з іншого пакету. Така ситуація сигналізує про погано спроектований дизайн.
- Діаграма класів має циклічну залежність між класами. Така ситуація сигналізує про погано спроектований дизайн.
- Діаграма класів має циклічну залежність між пакетами. Така ситуація сигналізує про погано спроектований дизайн [2, с. 480].

3.2. Протиріччя між діаграмою класів та діаграмою послідовностей

У діаграмі послідовностей відображається послідовність взаємодії між сутностями системи, що проявляється за допомогою повідомлень. Описані в діаграмі послідовностей відносини не повинні суперечити тому, як вони описанні в діаграмі класів, це важливо врегулювати тому можна винести такий список протиріч.

- Діаграма послідовностей використовує клас, що не описаний у діаграмі класів.
- При відправленні повідомлення в діаграмі послідовностей відповідний метод у класі-отримувачі повинен мати відповідний ідентифікатор доступу. У випадку якщо взаємодіють нащадок та батьківський клас, тоді метод може мати ідентифікатор доступу public або protected. Якщо класи не пов’язані такими залежностями між собою, то метод може бути тільки public.
- При відправленні повідомлення у діаграмі послідовностей використовується метод, якого не існує у відповідному класі-отримувачі, що описаний у діаграмі класів.
- Класу можуть бути відправлені повідомлення з використанням виключно статичних методів.
- Об’єкт може відправляти повідомлення з використанням тільки статичних методів, якщо між класами задана асоціація і також явно вказана відсутність доступу класу-відправника до класу-отримувача.
- Якщо в діаграмі класів задана асоціація між двома класами з кратністю один, то на діаграмі послідовностей не може об’єкт першого класу посилати повідомлення до двох різних об’єктів другого класу.
- Об’єкти класу зі стереотипом «utility» не можуть існувати, бо такий клас не може існувати [4, с.682].

3.3. Протиріччя між діаграмою класів та діаграмою об’єктів

Діаграма об’єктів призначена для демонстрації сукупності об'єктів, що моделюються, і зв'язків між ними у фіксований момент часу. По суті є екземпляром діаграми класів. Тому усі зв’язки між об’єктами не повинні суперечити тим, що вказані в діаграмі класів. Спираючись на це можна виділити такі протиріччя:

- Значення поля класу не сумісне з його типом, що вказаний у діаграмі класів.
- При композиції об’єкт-частина може належати тільки одному об’єкту-цілому одночасно.
- При асоціації двох класів то кількість екземплярів першого класу та кількість екземплярів другого не повинні суперечити тому, яка кількість регламентована у діаграмі класів.
- Об’єкти класу зі стереотипом «utility» не можуть існувати, бо такі класи не можна ініціалізувати [4, с.682].
- На діаграмі послідовностей об’єкти зв’язані, але цього зв’язку не має у діаграмі класів. Для того, щоб об’єкти могли бути зв’язані необхідно, щоб виконувалася хоча б одна з двох таких умов:
    - Між відповідними класами або їх батьківськими класами існує зв’язок.
    - У одного з відповідних класів або його батьківського класу є атрибут, тип якого співпадає з іншим класом або одним з його батьківським класом.
- Об’єкт класу, що має стереотип «implementationClass», не може бути екземпляром більш ніж одного класу. Те саме стосується і його нащадків [4, с.681].
- Для кожного поля об’єкту має виконуватись одне з таких правил:
    - існує асоціація між класом об’єкта або одним з його батьківських класів та деяким іншим класом. При цьому роль, що відображена при асоціації не суперечить імені атрибута.
    - існує атрибут з таким самим ім’ям у класу об’єкта або у одного з його батьківських класів.
- Якщо жодне з даних правил не виконується, це свідчить про наявність протиріч.

3.4. Протиріччя між діаграмою станів та іншими діаграмами

Діаграма станів описує ті стани об’єктів, які вони можуть досягати в період свого життєвого циклу. Найчастіше перехід з одного стану до іншого відбувається за допомогою виклику методів. Так як всі можливі методи, що мають класи описані в діаграмі класів, важливо щоб методи присутні у діаграмі станів не суперечили їм. Також послідовність виклику методів описана в діаграмі послідовностей і описане в діаграмі станів не повинно з цим суперечити. Далі будуть описані протиріччя, що випливають з даних тверджень.

- Послідовність відправлень повідомлень у діаграмі послідовностей не повинна суперечити множині послідовностей переходів у діаграмі станів даного класу.
- Кожен метод, що використовується для зміни стану об’єкта у діаграмі станів, повинен бути описаним у діаграмі класів.

3.5. Протиріччя всередині діаграми станів

- Кожен стан об’єкту повинен бути досяжним з початкового стану.
- Кінцевий стан завжди має бути досяжним.

4. Опис структури XML файлу

XML файл формату drawio зберігає інформацію про елементи діаграми та їхню форму. Але він не дає чіткого визначення елементам, наприклад що це клас або залежність. Але кожен елемент має ряд характеристик, що дає змогу зрозуміти що саме знаходиться в елементі. Надалі буде представлений детальний опис наявних у файлі тегів:

- *mxfile* – головний тег, у якості атрибутів має дату останнього редагування діаграми, версію застосунка та іншу технічну інформацію. Включає в себе теги diagram.
- *diagram* – це тег, який саме включає в себе діаграму. У якості атрибутів має id та назву діаграми. Включає у себе тег mxGraphModel.
- *mxGraphModel* – тег, що в атрибутах має інформацію про площину, на якій малюється діаграма, її розміри, тощо. Включає в себе тег root.
- *root* – це тег, що не має атрибутів, він включає в себе набір тегів mxCell.
- *mxCell* – це тег, що є основним в побудові діаграми. Усі її елементи, такі як: клас, метод, поле, залежність (якщо ми говоримо про діаграму класів), а також стрілки виклику функцій та значень, що повертаються (в контексті діаграм послідовностей). Тег mxCell може включати тег mxGeometry. Тег має такий список атрибутів:
  - *id* – унікальний ідентифікатор кожного елементу
  - *value* – текст елементу
  - *style* – рядок, що включає CSS стиль елементу
  - *parent* – це id елементу який на діаграмі включає у себе поточний елемент. Наприклад, parent елементу, що відображає метод буде id елементу, що відображає клас, який має цей метод.
  - *source* – наявний у стрілок, являє собою id елементу від якого прямує стрілка.
  - *target* – також специфічний для стрілок атрибут. Це id елемента до якого прямує стрілка.
  - *connectable* – атрибут наявний у описах до стрілок. Наприклад, при відображенні агрегації подібний елемент може зберігати інформацію про те скільки об’єктів буде агреговано.
- *mxGeometry* – тег, що відповідає за інформацію про те, скільки простору буде займати елемент або іншу геометричну інформацію. Має атрибути: X, Y, width, height. Може зберігати в собі теги mxRectangle або mxPoint.
- *mxRectangle* – являє собою опис прямокутника, як геометричної фігури на площині, має такі самі характеристики, як і тег mxGeometry. Зазвичай цей тег з’являється у класів.
- *mxPoint* – це опис точки, що може бути початком, кінцем, або точкою зламу для стрілки на діаграмі.

Опис відповідностей між тегами XML та елементами діаграми:

Усі згадані теги описують геометричні фігури, їх стиль та написи на них, а також чітко прив’язують одну фігуру до іншої за допомогою атрибутів parent, source та target. Це дає змогу виділити унікальність кожного елемента діаграми, а також класифікувати його. Далі розглянемо, які саме особливості дають змогу точно класифікувати елементи.

Почнемо з діаграми класів:

- *Клас* – як і усі елементи діаграми описується тегом mxCell. Особливість тегів mxCell, що описують саме класи це те, що всередині вкладеного тегу mxGeometry має ще тег mxRectangle. При цьому mxGeometry буде зберігати інформацію про розмір прямокутника, у якому записана лише назва класу, а mxRectangle - розміри усього класу. Це обумовлено тим, що від класу до класу прямують стрілки залежностей, і чіткий опис границь необхідний для встановлення до яких саме класів вони належать.

- *Поля класу* – також описані тегами mxCell. Тег root містить у собі список тегів mxCell, при чому ці теги впорядковані таким чином, що всі поля та методи класу записані безпосередньо після тегу самого класу. Також вони обов'язково мають в атрибуті 'parent' id класу, якому належить. Також усі поля класу – це наступні теги mxCell після тегу самого класу.
- *Методи класу* – усі методи класу – це наступні теги mxCell після полів класу та одного тегу mxCell з порожнім атрибутом value. На діаграмі – це роздільна лінія між полями та методами класу. mxCell методу має в атрибуті 'parent' id класу, якому належить. У разі, якщо клас не має полів, усе одно буде присутній порожній простір і лінія, що відмежовує назву класу та його методи на діаграмі. А в XML файлі це означає, що mxCell з порожнім атрибутом value збережеться.
- *Стрілка залежності* – це ті теги mxCell, має атрибути source та target. Для різних типів залежностей стрілка може описуватися одним тегом mxCell або кількома. Наприклад, якщо це агрегація або композиція, то буде використаний один тег та у атрибуті value буде зазначено кількість залежних класів. Наслідування та реалізація мають порожній рядок в атрибуті value, з тою відмінністю, що реалізація у атрибуті style має рядок 'dashed=1', тобто пунктирна лінія.

Діаграма послідовностей також відображає класи, методи та залежності, але в інший спосіб. Опис класифікації елементів діаграми послідовностей:

- *Клас* – особливість класів у діаграмі послідовностей – це наявність лінії, від якої йдуть виклики методів у хронологічному порядку. Даний сервіс малює також прямокутник на лінії, саме від нього йдуть виклики, тобто його можна вважати частиною класу. В цьому випадку найпростіший спосіб розпізнати клас – це знайти mxCell тег, що в атрибуті style буде мати рядок 'perimeter=lifelinePerimeter', це буде сам клас. А також mxCell, що має в цьому атрибуті рядок 'perimeter=orthogonalPerimeter', а в атрибуті parent id класу – це прямокутник, що належить класу. Його важливо запам'ятовувати, бо стрілка виклику функції у початку або в кінці може прямувати саме до прямокутника і мати у source або target id саме прямокутника.
- *Метод* – відображається стрілкою виклику методу. Може бути заданий двома чи однією стрілкою. Перша містить інформацію про назву методу та набір аргументів. Друга – про значення, що повертається. Другої стрілки може не бути у випадках, коли метод нічого не повертає. Головною їх відмінністю є те, що стрілка значення, що повертається, завжди є пунктирною. За цією особливістю можна її розпізнати, воно у атрибуті style містить рядок 'dashed=1'. А на основі того, до якого класу прямує стрілка виклику, можна зробити висновок якому класу належить метод. На відміну від методів, які класифікуються з діаграми класів, методи з діаграми послідовностей будуть зберігати інформацію про класи, що їх викликають. Ця інформація буде корисна у подальшому пошуку протиріч.
- *Залежності* – вони не задаються явно і не можуть бути чітко встановленими. Головне – це те, що якщо об'єкт класу може викликати метод іншого класу тільки якщо він має якусь залежність із ним.

3. Методи верифікації цілісності

Головною метою дослідження є написання програми, що буде розпізнавати протиріччя в межах однієї діаграми та між двома діаграмами.

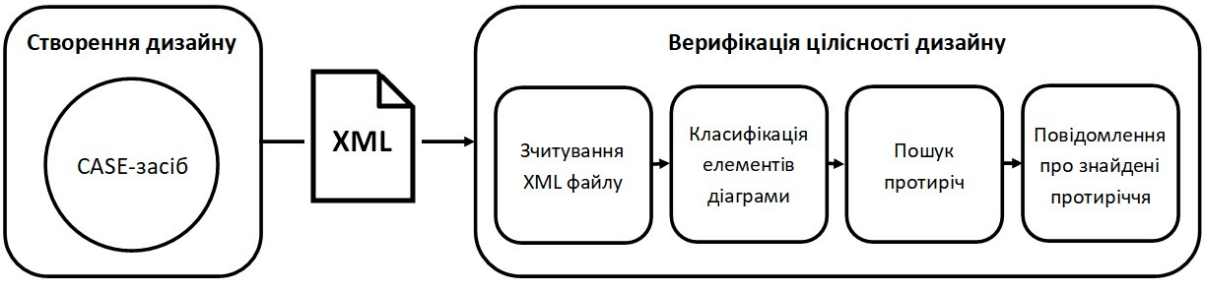

Рис.1 - Загальна схема пошуку протиріч.
Fig.1 - General schema of finding conflicts.

Результатом парсингу XML файлів діаграм класів та послідовностей є списки об’єктів, що описують класи, представленні у діаграмах.Далі будуть описані методи пошуку конкретних протиріч:

1. *У діаграмі послідовностей є клас, що не описаний у діаграмі класів.* Маючи список класів з діаграми класів, порівнюємо його зі списком класів з діаграми послідовностей. Якщо другий список містить класи, яких немає у першому, то це порушення цілісності.

2. *Нелегітимне використання методів класу.* Виклики методів у діаграмі послідовностей повинні узгоджуватись з діаграмою класів. Можливість використання методів інших класів залежить від багатьох факторів, кожен із яких має бути перевіреним. Надалі буде представлений опис перевірки кожного фактору:

2.1. *Перевірка чи є відповідний метод у класі, що викликається.* У метода є три характерні особливості, що для програми роблять його унікальним – це значення, що він повертає, назва та список аргументів. Перевіряти потрібно всі, отже:

2.1.1. *Назва.* Перевірка назви буде виконуватись першою і перевіряти чи є хоч один метод з такою назвою. Перевірка робиться рекурсивно для всіх батьківських класів, з модифікаторами доступу public, default та protected.

2.1.2. *Список аргументів.* З попереднього кроку формується список методів, що мають потрібну назву. Далі у цьому списку шукаються методи з необхідними списками аргументів.

2.1.3. *Значення, що повертається.* У списку класів, що сформований на попередніх кроках, тобто з потрібною назвою та списком аргументів, шукаємо метод з відповідним значенням, що повертається.

2.2. *Перевірка на легітимність виклику методу.* Після того, як метод, що викликається у діаграмі послідовностей, знайдений у діаграмі класів, перевіряємо його рівень доступу. Якщо private, то легітимним буде тільки його виклик цим же класом. Якщо protected, то рекурсивно перевіряються ієрархія класу, що містить цей метод. Якщо клас, що його викликає, не є його батьківським – це порушення цілісності. За назвою пакету перевіряється default. Для public перевірка не проводиться.

3. *Приналежність об’єкта-частини до кількох об’єктів-цілих при композиції.* Це протиріччя можна виявити в двох ситуаціях:

3.1. *Кратність композиції більше 1.* Композиція в діаграмі класів має позначку кратності. Після парсингу XML файлу ця інформація зберігається. Для перевірки використовується регулярний вираз, що знайде композицію, у якій вказана кратність більша за одиницю.

3.2. *Об’єкт є частиною більш ніж однієї композиції.* Для пошуку береться список усіх композицій в діаграмі класів. З цього списку утворюємо список класів, що є залежними в цих композиціях. Далі перевіряється чи містить знайдений список повторення.

4. *Циклічна залежність класів.* Діаграма класів представляється у вигляді орієнтованого графа, у якому вершини – це класи, а ребра – залежності. Далі проводиться пошук циклів у орієнтованому графі.

5. *Несумісні стереотипи класу.* Після парсингу XML файлу інформація про стереотипи зберігається. Програма, що перевіряє містить інформацію, про стереотипи, що не є сумісними. Тож для всіх класів перевіряється список стереотипів.

6. Застосунок на мові програмування Java, що реалізує описані методи пошуку протиріч

У ході роботи була створена програма, що перевіряє наявність протиріч у двох типах UML діаграм об'єктно-орієнтованого програмування: діаграмі класів та діаграмі послідовностей, та між ними. Програма написана на мові програмування Java. Програмний код розділено на 5 пакетів на основі зон відповідальності класів, які вони містять. Послідовність використання цих пакетів зображено на діаграмі.

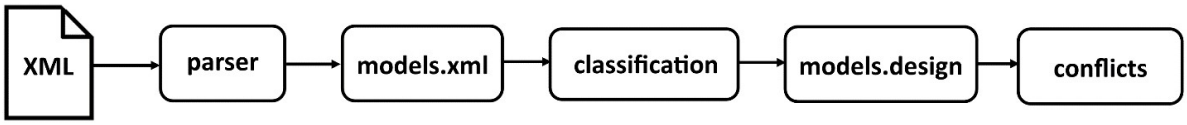

Рис.2 - Послідовність використання програмних пакетів.
Fig.2 - The sequence of program packages usage.

Далі розберемо як саме програма реалізує перевірку цілісності об'єктно-орієнтованого дизайну:

1. *Зчитування XML файлу, що описує діаграму, та представлення її у виді сукупності об'єктів різних типів у середині програми.*

Опис сутностей, створених у програмі для відображення сутностей XML файлу. Те, які теги та атрибути має XML файл діаграм було описано вище. Для зчитування і зберігання даних з файлу були створені спеціальні класи у програмі. Ці класи знаходяться у пакеті **models.xml**.

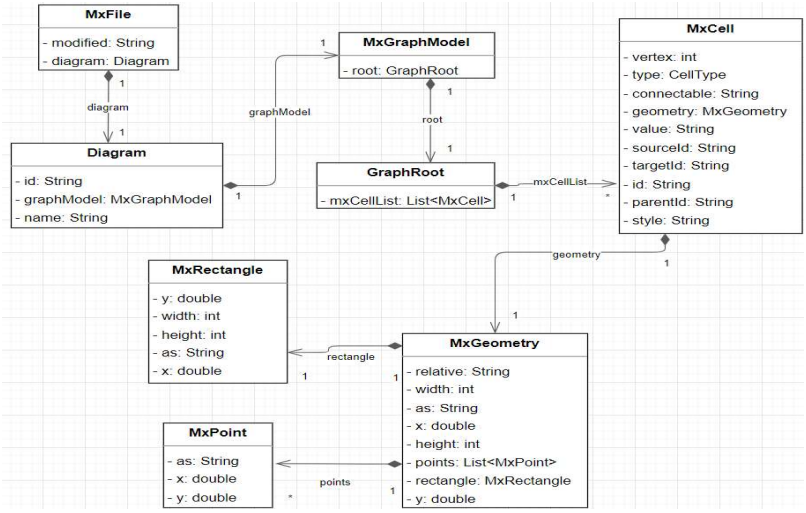

Рис.3 - Діаграма класів пакету **models.xml**.
Fig.3 - **models.xml** package class diagram.

Опис сутностей, що відображають елементи UML діаграм класів та послідовностей у програмі. Класи, що зберігаються в пакеті **models.xml**, не відображають UML діаграму у зрозумілому для об’єктно-орієнтованого програмування вигляді. Вони скоріш описують те, як малювати ту чи іншу діаграму. Сама ж UML діаграма містить такі сутності, як, наприклад, клас, метод, чи залежність. Тож для ефективного пошуку протиріч програма повинна зберігати діаграму саме в таких сутностях. Для цього були створені спеціальні класи, що зберігаються у пакеті **models.design**.

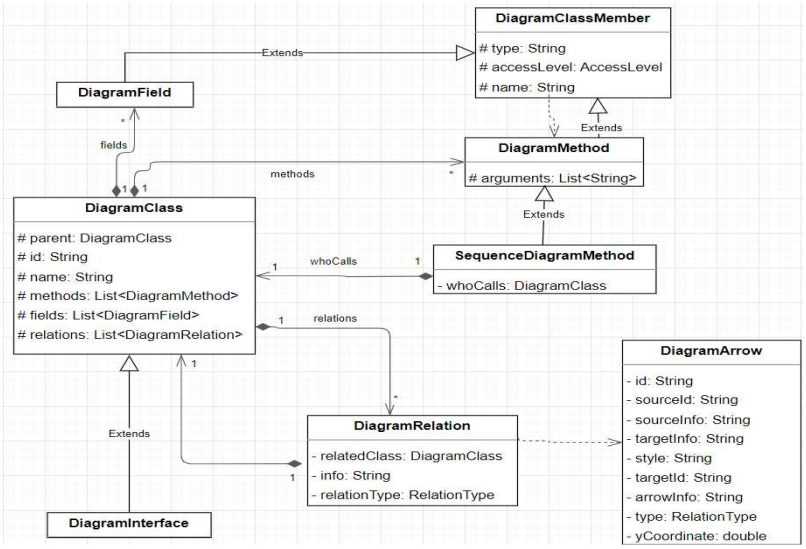

Рис.4 - Діаграма класів пакету **models.design**.
Fig.4 - **models.design** package class diagram.

Зчитування та парсинг XML файлу. Тепер, коли програмамає класи, у які буде записуватися інформація про діаграму, її можна зчитувати з XML файлу. Для зчитування та парсингу XML була використана бібліотека **org.xml.sax**. Результатом парсингу буде набір об’єктів класів з пакету **models.xml**. Класи, що займаються парсингом знаходяться у пакеті **parser**.

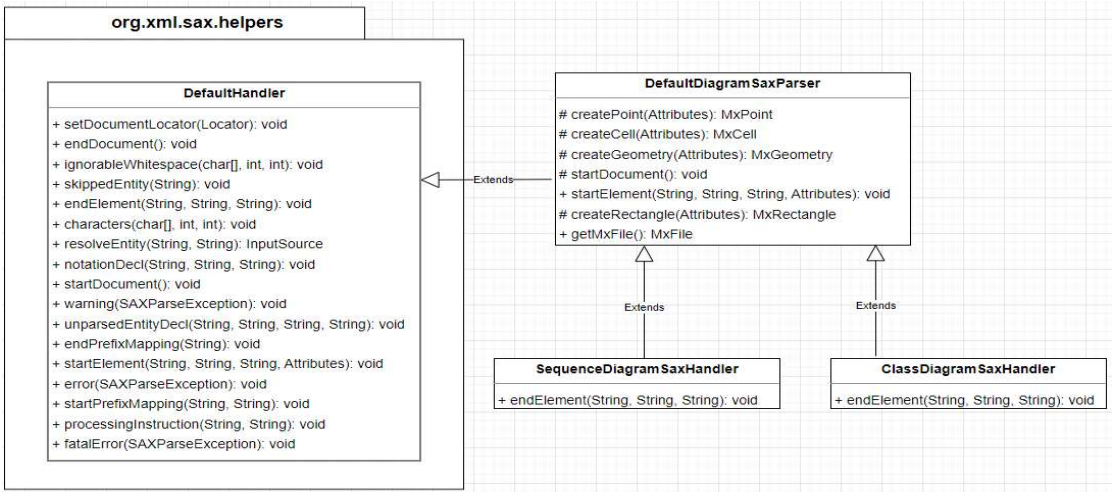

Рис.5 - Діаграма класів пакету **parser**.
Fig.5 - **parser** package class diagram.

Класифікація елементів UML діаграм. Наступна задача з набору об’єктів класів з пакету **models.xml**, що були отримані в минулому етапі класифікувати саме елементи діаграми, тобто класи, методи, поля, залежності, тощо. Для цього потрібно якимось чином створити об’єкти класів з пакету **models.design**. З цією метою було створено пакет класів **classification**.

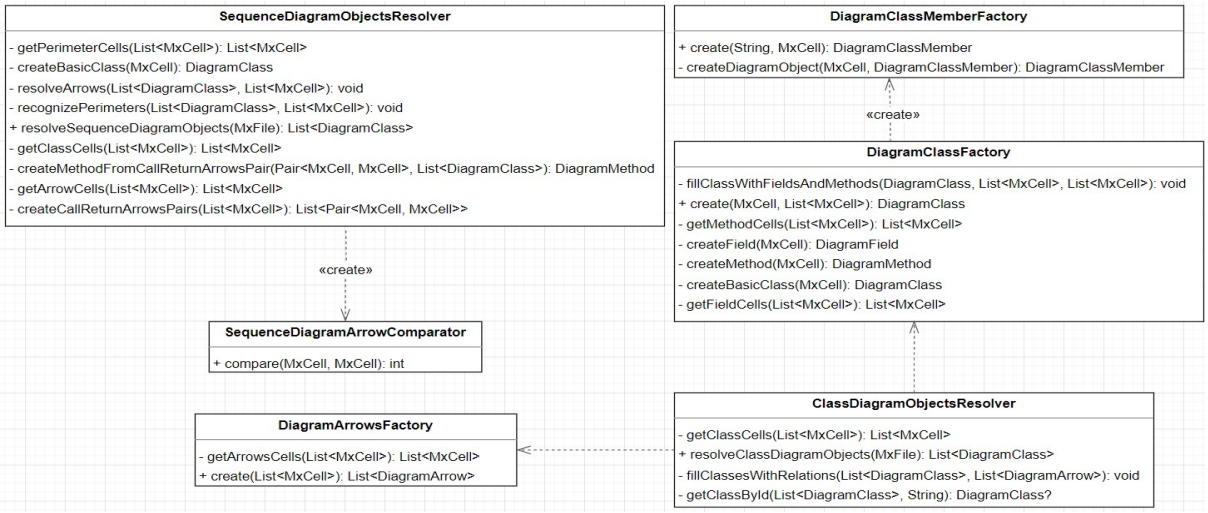

Рис.6 - Діаграма класів пакету **classification**.
Fig.6 - **classification** package class diagram.

2. *Пошук протиріч.*

Останнім і найголовнішим етапом буде пошук протиріч. Після класифікації, дані діаграми знаходяться у зручному для їх обробки вигляді. Усі сутності в програмі відповідають дійсним сутностям діаграм UML. Усі класи, що займаються пошуком протиріч знаходяться в пакеті **conflicts**.

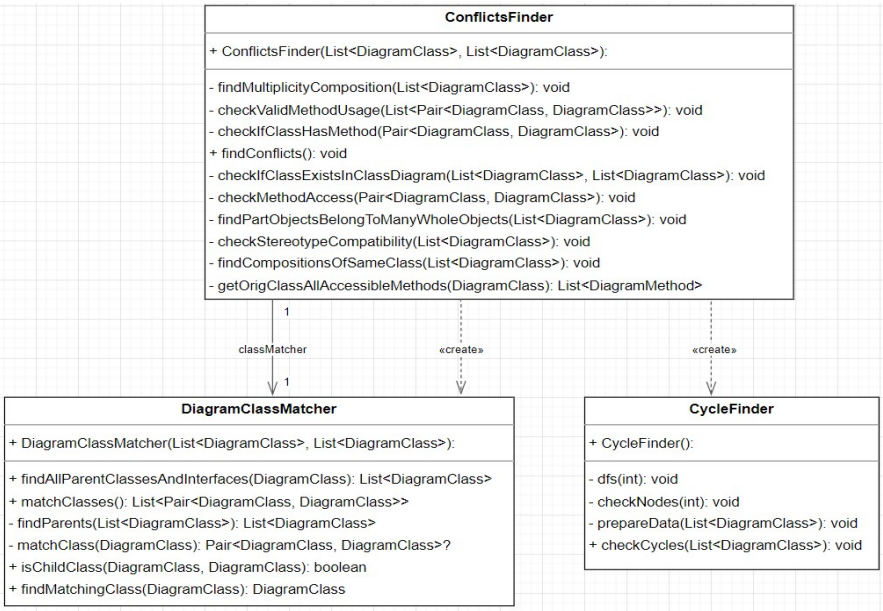

Рис.7 - Діаграма класів пакету **conflicts**.
Fig.7 - **conflicts** package class diagram.

7. Висновки

У результаті проведеного дослідження було виявлено близько двадцяти протиріч, що можуть виникати у об’єктно-орієнтованому дизайні. Були розроблені методи знаходження та виявлення протиріч. А також реалізовано додаток на мові програмування Java, який успішно реалізує виведені протиріччя.

Перспективою розвитку даної роботи можна вказати наступне. По-перше, виявлення ще більшої кількості потенційних протиріч у діаграмах об’єктно орієнтованого дизайну. По-друге, є дуже великі перспективи по масштабуванню та удосконаленню застосунку для пошуку протиріч. Є дуже багато варіантів його покращення, починаючи з удосконалення алгоритмів, закінчуючи універсальністю вхідних даних, тобто варіантів представлення діаграм, що подаються на вхід до програми, та вдосконаленням користувацького досвіду.

Завершуючи, можна сказати, що результати цього дослідження сприятимуть покращенню процесу проектування програмного забезпечення та розвитку даної галузі в цілому.

Конфлікт інтересів

Автори повідомляють про відсутність конфлікту інтересів.

References

1. Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, & Jim Conallen. (2007). *Object-Oriented Analysis and Design with Applications* (3rd ed.). https://zjnu2017.github.io/OOAD/reading/Object.Oriented.Analysis.and.Design.with.Applications.3rd.Edition.by.Booch.pdf
2. Craig Larman. (2004). Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. https://bsituos.weebly.com/uploads/2/5/2/5/25253721/applying-uml-and-patterns-3rd.pdf
3. Vanessa Weber, Kleinner Farias, Lucian Gonçales & Vinícius Bischoff. (2016). Detecting Inconsistencies in Multi-view UML Models. *International Journal of Computer Science and Software Engineering (IJCSSE)*, Volume 5, Issue 12. https://www.researchgate.net/publication/313837603_Detecting_Inconsistencies_in_Multi-view_UML_Models
4. OMG. Unified Modeling Language 2.5.1 Specification. (2017). https://www.omg.org/spec/UML/2.5.1/
5. Gamma Erich, Helm Richard, Johnson Ralph & Vlissides John. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. https://www.javier8a.com/itc/bd1/articulo.pdf
6. Robert C. Martin. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. https://ptgmedia.pearsoncmg.com/images/9780132928472/samplepages/0132928477.pdf
7. Martin Fowler, Kent Beck, John Brant, William Opdyke & Don Roberts. (1999). Refactoring: Improving the Design of Existing Code. https://ptgmedia.pearsoncmg.com/images/9780201485677/samplepages/9780201485677.pdf
8. Joyce Farrell. (2017). Programming Logic and Design, Introductory. https://jamborebook.co/download/4867679-program-logic-and-design
9. Iryna Zaretska, Oleksandra Kulankhina & Hlib Mykhailenko. Cross-Diagram UML Design Verification. *ICT in Education, Research and Industrial Applications. CCIS*, Vol. 347, Springer-Verlag, Berlin Heidelberg (2013). – pp. 165-176. http://dx.doi.org/10.1007/978-3-642-35737-4_10
10. Iryna Zaretska, Oleksandra Kulankhina, Hlib Mykhailenko & Tamara Butenko. Consistency of UML Design. *International Journal of Information Technology and Computer Science (IJITCS)*, Vol.10, No.9, 2018. – pp. 47-56. https://doi.org/10.5815/ijitcs.2018.09.06
11. Rational Rose. https://www.ibm.com/docs/en/rational-clearquest/7.1.0?topic=developing-schemas-clearquest-designer
12. Diagrams.Net. https://app.diagrams.net/

DEVELOPMENT AND IMPLEMENTATION OF A METHOD FOR CHECKING THE INTEGRITY OF THE DESIGN OF AN OBJECT-ORIENTED SYSTEM

Mykyta Pugach[1], PhD student, Department of Theoretical and Applied Informatics, Faculty of Mathematics and Informatics; e-mail: nikita.pugach.2000@gmail.com;
ORCID: https://orcid.org/0009-0004-8923-6489
Iryna Zaretska[1], associate professor of the institution of higher education, Department of Theoretical and Applied Informatics, Faculty of Mathematics and Informatics; e-mail: zaretskaya@karazin.ua;
ORCID: https://orcid.org/0000-0001-8747-2737
[1] *V. N. Karazin Kharkiv National University, Ukraine*
Manuscript was received April 3, 2024; Received after review May 5, 2024; Accepted June 7, 2024

Abstract. Creating modern software products is a complex and long process consisting of many parts. To achieve quality, it is necessary to carry out various measures for testing and verifying software at all stages of development. This article discusses the software design stage, which is integral and one of the most important. In most software development life cycle (SDLC) models, this stage is one of the first, so design mistakes will lead to problems in all subsequent stages. Thus, due to the high cost of error, it is very important to check the integrity of the developed design at the design stage. The article examines the problem of finding contradictions in object-oriented design. The authors present a set of contradictions that can arise in such a design and aim to develop methods and algorithms for detecting and searching for these contradictions in order to improve the quality of the design, as well as writing software that will implement these algorithms and methods. The program "diagrams.net" was chosen as a tool for creating object-oriented design, the main useful feature of which is the ability to present the created diagrams in the form of an XML file in the popular drawio format. The authors of the study propose a method for parsing the XML file of the diagram and presenting it as a set of objects, such as dependency arrows, classes, methods, etc. These objects must interact according to the established rules.The violation of these rules is a contradiction of the object-oriented design. As a result of the study, a method of finding contradictions was presented and implemented in the Java programming language.

Keywords: *software development, UML, software design, object-oriented design diagrams*

Conflicts of Interest: the authors declare no conflict of interest.