Харківський національний університет імені В.Н. Каразіна

механіко-математичній факультет

кафедра теоретичної та прикладної механіки

ДИПЛОМНА РОБОТА

магістра

на тему «ОБ’ЄКТНО-ОРІЄНТОВАНИЙ ДИЗАЙН ТА ШАБЛОНИ. ВЕРИФІКАЦІЯ СУМІСНОСТІ ТА ЦІЛІСНОСТІ ДИЗАЙНУ»

Виконав:
студент 6 курсу, групи МФ-61
спеціальність 122 «Комп’ютерні науки»
освітньо-професійна програма
«Інформатика»
Пугач М.С.

Керівник: доцент кафедри теоретичної та
прикладної інформатики
к.ф-м.н. Зарецька І.Т.

Харків-2023

ЗМІСТ

1. ВСТУП **3**

1.1. Формулювання мети, задач та обґрунтування актуальності теми 3

1.2. Огляд існуючих рішень 5

1.3. Відомості про одержані результати та їх новизна 6

1.4. Теоретичне та практичне значення результатів 7

2. ОСНОВНА ЧАСТИНА..**8**

2.1. Опис протиріч 8

2.1.1. Протиріччя всередині діаграми класів або між діаграмами класів 8

2.1.2. Протиріччя між діаграмою класів та діаграмою послідовностей 12

2.1.3. Протиріччя між діаграмою класів та діаграмою об'єктів............16

2.1.4. Протиріччя між діаграмою станів та іншими діаграмами...........21

2.1.5. Протиріччя всередині діаграми станів.......................................22

2.2. Опис застосунку для створення UML діаграм 23

2.3. Опис методів пошуку протиріч 28

2.4. Опис застосунку на мові програмування Java, що реалізує описані методи пошуку протиріч 35

3. ВИСНОВКИ **47**

4. СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ **48**

1. ВСТУП

1.1. Формулювання мети, задач та обґрунтування актуальності теми

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, в слід 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.

Мета даного дослідження це розробка методів та алгоритмів для виявлення та пошуку протиріч в об’єктно орієнтованому дизайні з метою покращення якості проектування, а також написання програмного забезпечення, що буде реалізовувати дані алгоритми та методи.

Для досягнення мети були поставленні такі задачі:

- Дослідження наукової літератури на тему об'єктно орієнтованого дизайну та уніфікованої мови моделювання UML на предмет опису потенційних несумісностей та протиріч у дизайні.
- Дослідження найновішої специфікації уніфікованої мови моделювання UML 2.5.1.
- Аналіз існуючих рішень з пошуку протиріч, серед яких можуть бути Critic або Case засоби.
- Складання списку виявлення можливих протиріч у дизайні систем.
- Розробка методів та алгоритмів пошуку протиріч у діаграмах UML.
- Реалізація цих методів та алгоритмів пошуку протиріч у виді програмного забезпечення.

Основна ціль даного дослідження полягає у глибинному вивченні можливих недоліків, протиріч, що можуть виникати на етапі проектування програмного забезпечення. Окрім того робота полягає у формуванні методів та алгоритмів для виявлення та пошуку таких протиріч з метою

У роботі 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 буде стосуватися пошуку протиріч та несумісності саме в цих чотирьох діаграмах та між ними.

Актуальність даного дослідження полягає саме в тому, що етап проектування є одним з найважливіших етапів розробки програмного забезпечення. Помилки на цьому етапі коштуватимуть найдорожче серед усіх інших етапів. А представлена робота ставить за ціль мінімізувати ці помилки та зробити перевірку дизайну системи зручнішим та швидшим.

1.2. Огляд існуючих рішень

На даний момент у світі вже існують засоби, що допомагають знайти деякі протиріччя у дизайні систем з використанням уніфікованої мови моделювання UML. Наприклад, одним із різновидів таких програмних рішень є CASE-засоби. CASE-засоби це абревіатурою від «Computer-Aided Software Engineering». CASE-засоби є програмними інструментами, призначеними для підтримки різних аспектів розробки програмного забезпечення, включаючи аналіз, проектування, розробку, тестування та управління проектом. Дані засоби надають деякі інструменти, що допомагають з проектуванням програмного продукту. Серед можливостей є моделювання, аналіз і верифікація, кодування та управління проектом. Для цілей, що пересікаються з даною роботою можна виділити зокрема аналіз та верифікація. CASE-засоби надають можливість аналізувати моделі та проводити пошуку та виявлення протиріч, перевірки правильності моделювання та дотримання правил моделювання.

Одним з провідних і найфункціональніших представників CASE-засобів є Rational Rose. Це інструмент для моделювання, що дозволяє створювати UML-діаграми, включаючи діаграми класів, послідовностей та компонентів. Rational Rose забезпечує можливості аналізу та перевірки консистентності моделей.

Саме останній компонент і представляє найбільший інтерес. Даний програмний продукт дійсно має можливості вирішувати деякі з протиріч, що будуть розглянуті в даній роботі. Але його можливості не повні і не вирішують весь спектр проблем з протиріччями та неконсистентністтю, які можуть виникнути на етапі проектування. Це зумовлено тим, що CASE-засоби спираються тільки на специфікацію мови моделювання UML, але насправді

сама UML дає набагато більше можливостей ніж специфіковано групою OMG (Object Management Group), некомерційною організацією, що займається розробкою та підтримкою різноманітних стандартів у моделюванні програмного забезпечення, зокрема UML. І це саме зумовлює можливість виникнення протиріч, які не специфікуються, і, як наслідок, такі протиріччя не можуть бути знайденими CASE-засобами, зокрема, Rational Rose.

На рисунку 1 буде наведено приклад інтерфейсу та пошуку протиріччя програмою Rational Rose**.**

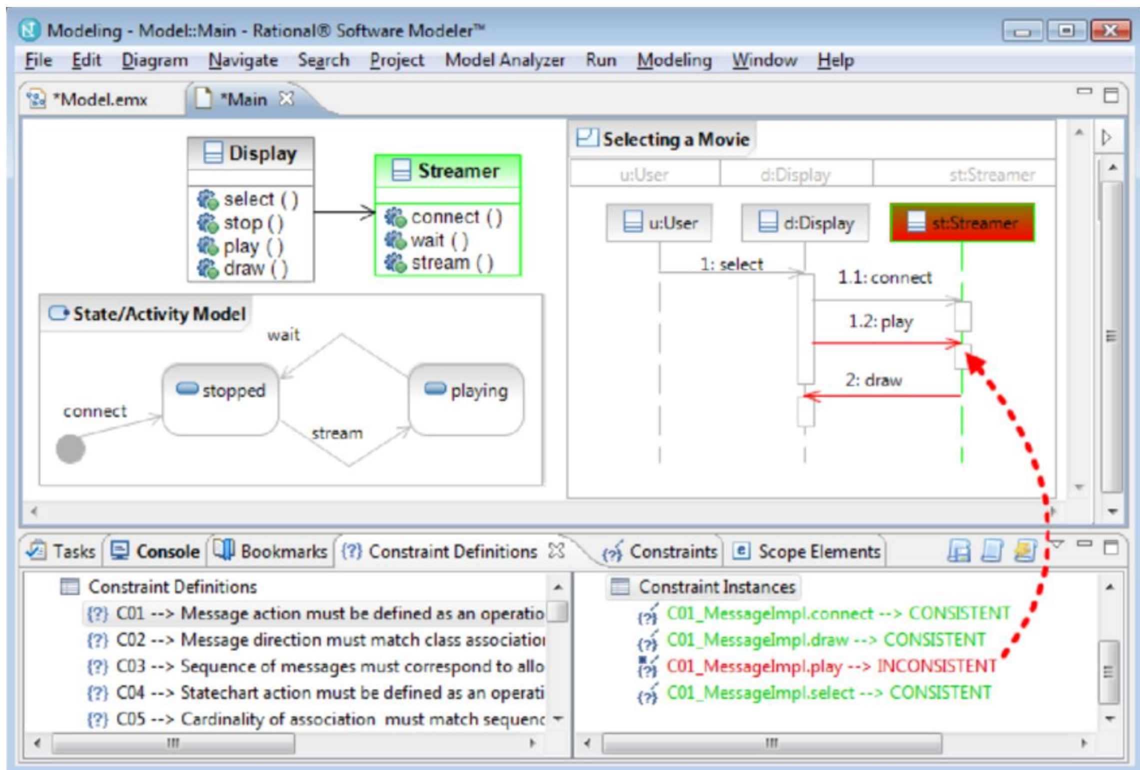

Рис. 1. Приклад роботи програми Rational Rose.

1.3. Відомості про одержані результати та їх новизна

У результаті написання даної роботи були одержані такі результати:

- Проаналізовано потенційні уразливі місця у діаграмах об'єктно орієнтованого дизайну і знайдено близько двадцяти протиріч, що

можуть виникати всередині чи на перетині чотирьох різновидів діаграм. А саме: діаграми класів, послідовностей, станів та об’єктів.
- Сформульовано методи та алгоритми пошуку таких протиріч.
- Реалізовано програму на мові програмування Java, що використовує засіб для створення «diagrams.net» для експорту звідти діаграм та реалізовує дані методи та алгоритми, і в результаті знаходить протиріччя у UML діаграмах об’єктно орієнтованого дизайну.

Новизною даної роботи в основному є те, що реалізована програма об’єднує в собі находження протиріч, не всі з яких можуть бути знайдені існуючими засобами. А також, на відміну тих засобів, що є зараз, дана програма є окремою від застосунків для створення UML діаграм. На даний момент вона працює тільки з одним застосунком, але в перспективі це може бути будь-який. Це означає, що розробити діаграми можна будь-де, а потім перевірити за допомогою програми, створеної в результаті цієї роботи.

1.4. Теоретичне та практичне значення результатів

Важливим результатом роботи з теоретичної точки зору було формулювання протиріч у UML діаграмах об’єктно орієнтованого дизайну. А також розробка методів та алгоритмів їх знаходження, описано як, використовуючи відомості про елементи різних типів діаграм, знаходити протиріччя.

Практичним результатом є створення програми на мові програмування Java, що реалізує сформульовані методи та алгоритми. При подальшому розвитку і масштабуванні вона здатна серйозно конкурувати з існуючими розробками в цьому напрямку, з огляду, на свою новизну, що була описана раніше.

2. ОСНОВНА ЧАСТИНА

2.1. Опис протиріч

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 в середині одної 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.

2.1.1. 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:

Протиріччя 1

Клас має два або більше стереотипи, що не є сумісними між собою. Наприклад, клас одночасно позначений стереотипами «exception» та

«enumeration».

Приклад протиріччя забражений на рисунку 2.

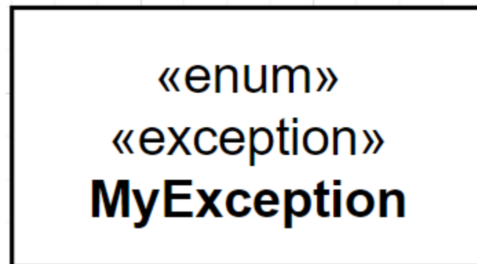

Рис.2. Приклад протиріччя: клас має два несумісних стереотипи.

Протиріччя 2

У випадку композиції кожен об’єкт-частина може належати тільки одному об’єкту-цілому. Для того, щоб ця вимога дотримувалася у діаграмі повині бути враховані такі вимоги:

- Кратність кінця-цілого при композиції не повинна перевищувати одиниці;
- Один клас не може бути частиною більше ніж однієї композиції.

Приклади протиріч зображені на рисунках 3 та 4.

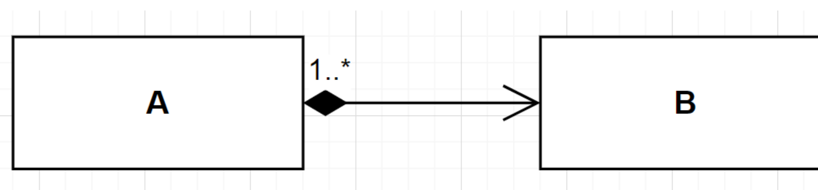

Рис.3. Приклад протиріччя: кратність композиції більше одиниці.

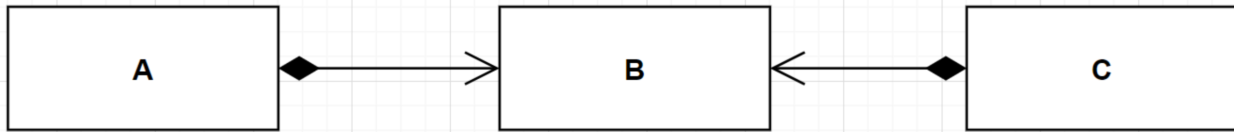

Рис.4. Приклад протиріччя: Об'єкт є частиною двох композицій.

Протиріччя 3

Клас, що має стереотип «utility», не може містити у собі методи або атрибути з областю видимості екземпляру [6, с.682].

Приклад протиріччя зображений на рисунку 5.

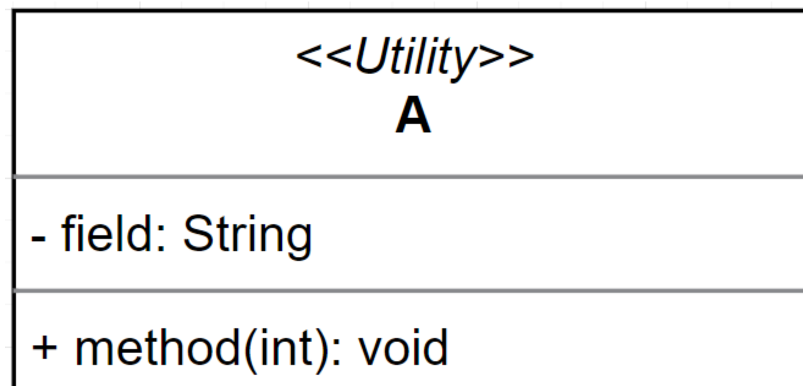

Рис.5. Приклад протиріччя: клас зі стереотипом «utility» містить атрибут із областю видимості 'екземпляр'.

Протиріччя 3

Клас успадковує клас з іншого пакету. Така ситуація сигналізує про погано спроектований дизайн.

Приклад протиріччя зображений на рисунку 6.

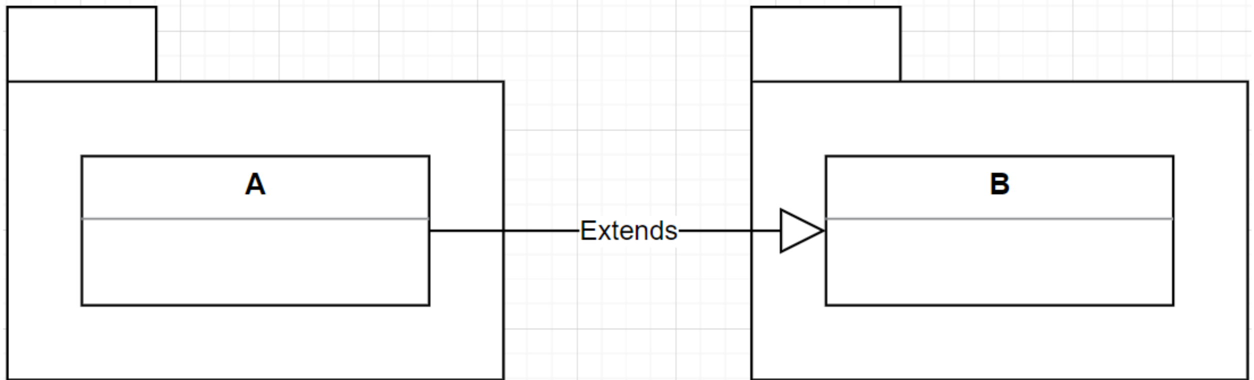

Рис.6. Приклад протиріччя: наслідування класів, що знаходяться у різних пакетах.

Протиріччя 4

Діаграма класів має циклічну залежність між класами. Така ситуація сигналізує про погано спроектований дизайн.

Приклад протиріччя зображений на рисунку 7.

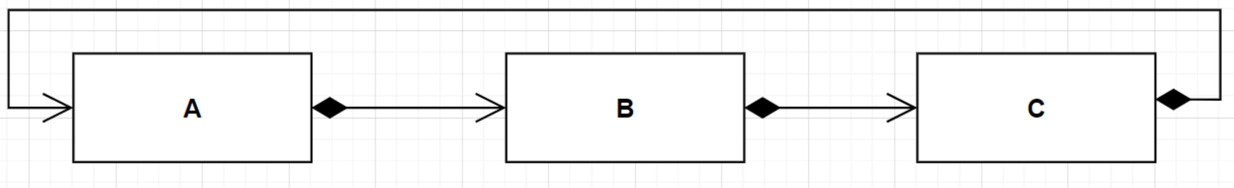

Рис.7. Приклад протиріччя: циклічна залежність класів.

Протиріччя 5

Діаграма класів має циклічну залежність між пакетами. Така ситуація сигналізує про погано спроектований дизайн [4, с. 480].

Приклад протиріччя зображений на рисунку 8.

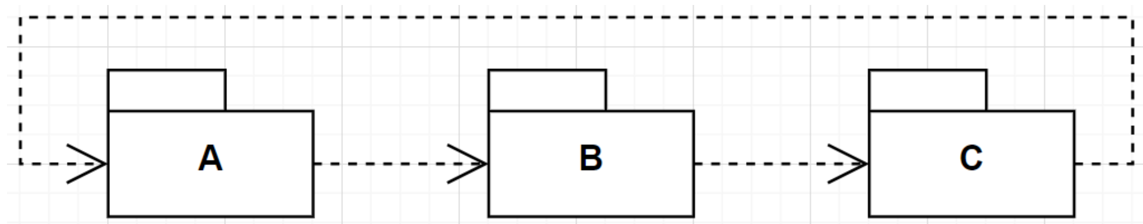

Рис.8. Приклад протиріччя: циклічна залежність пакетів.

2.1.2. Протиріччя між діаграмою класів та діаграмою послідовностей

Діаграма послідовностей створюється для чіткого врегулювання відносин між класами. У ній 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.

Протиріччя 6

Діаграма послідовностей використовує клас, що не описаний у діаграмі класів.

Приклад протиріччя зображено на рисунку 9.

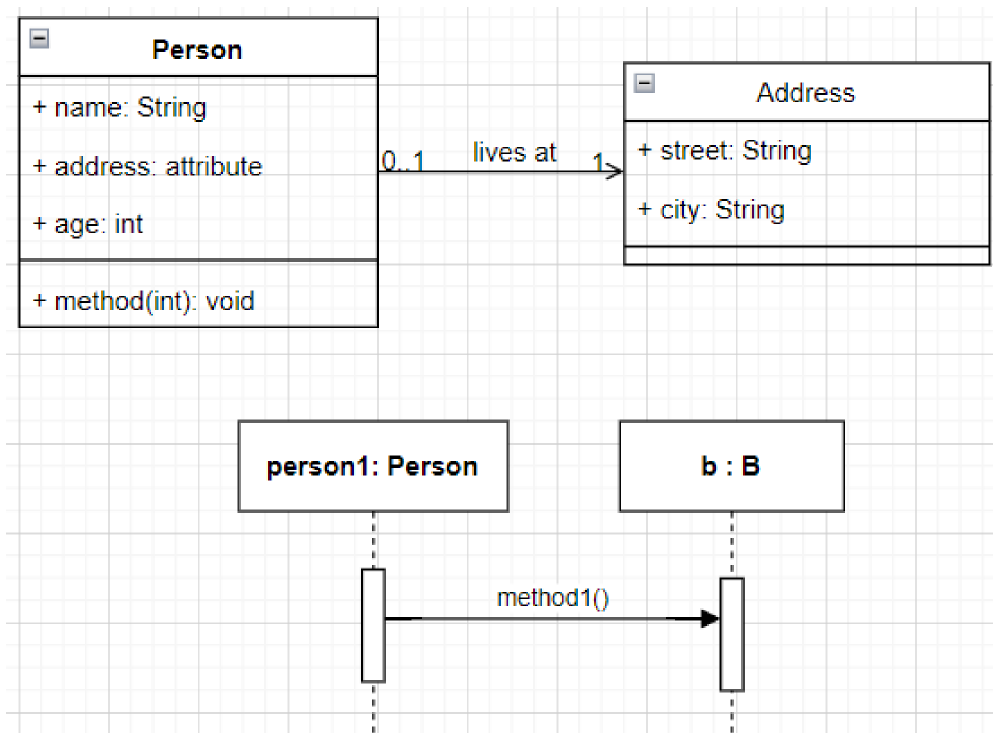

Рис.9. Приклад протиріччя: діаграма послідовностей містить неіснуючий у діаграмі класів клас.

Протиріччя 7

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*. Приклад протиріччя зображений на рисунку 10.

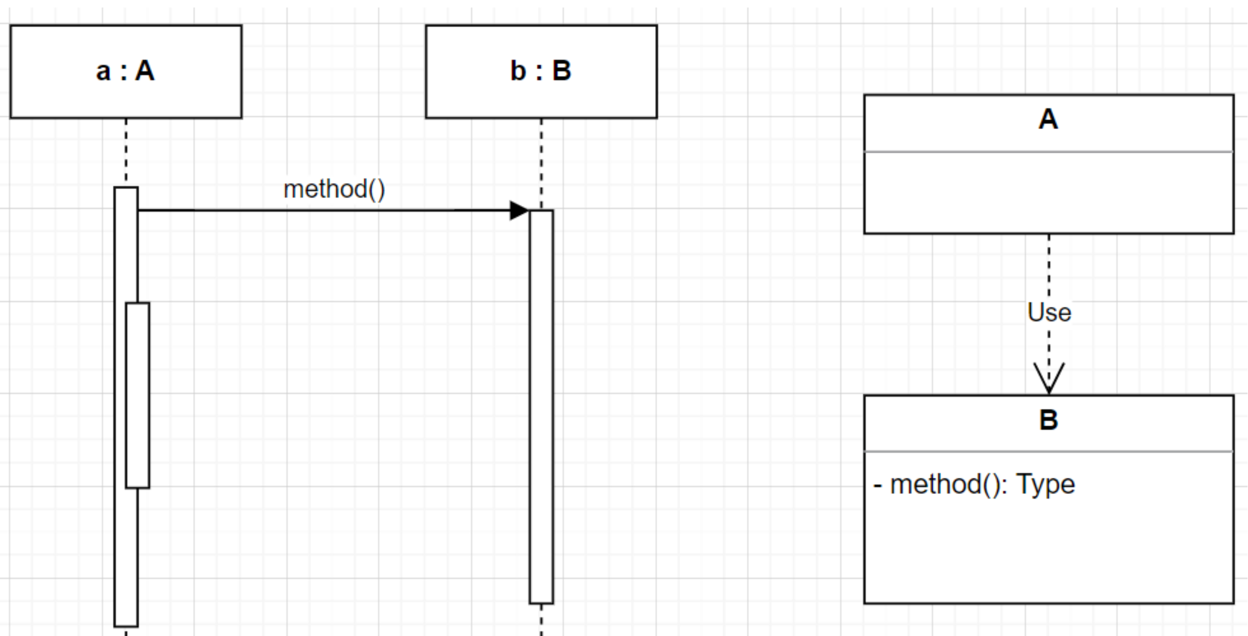

Рис.10. Приклад протиріччя: виклик private методу.

Протиріччя 8

При відправленні повідомлення у діаграмі послідовностей використовується метод, якого не існує у відповідному класі-отримувачі, що описаний у діаграмі класів.

Приклад протиріччя зображений на рисунку 11.

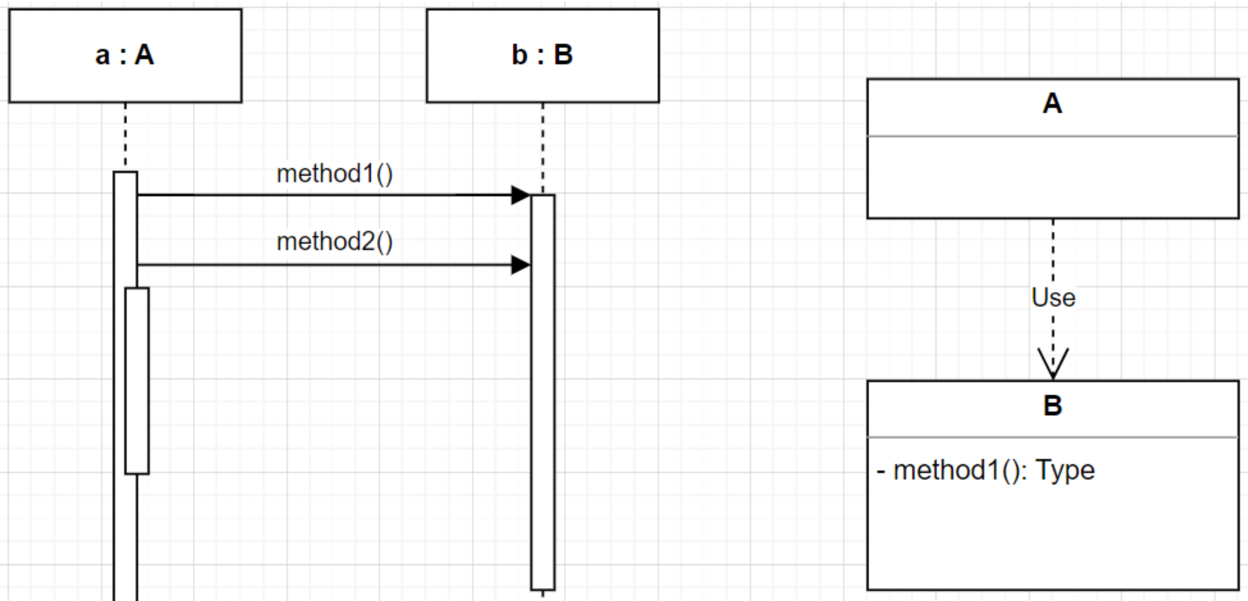

Рис.11. Приклад протиріччя: виклик методу, якого не існує у відповідному класі.

Протиріччя 9

Класу можуть бути відправлені повідомлення з використанням виключно статичних методів.

Приклад протиріччя зображений на рисунку 12.

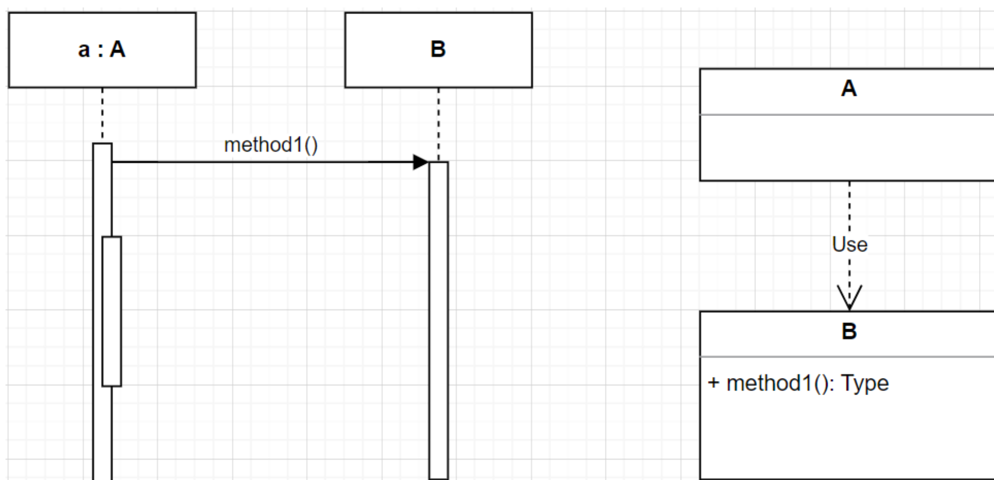

Рис.12. Приклад протиріччя: виклик не статичного методу у класу.

Протиріччя 10

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.

Приклад протиріччя зображений на рисунку 13.

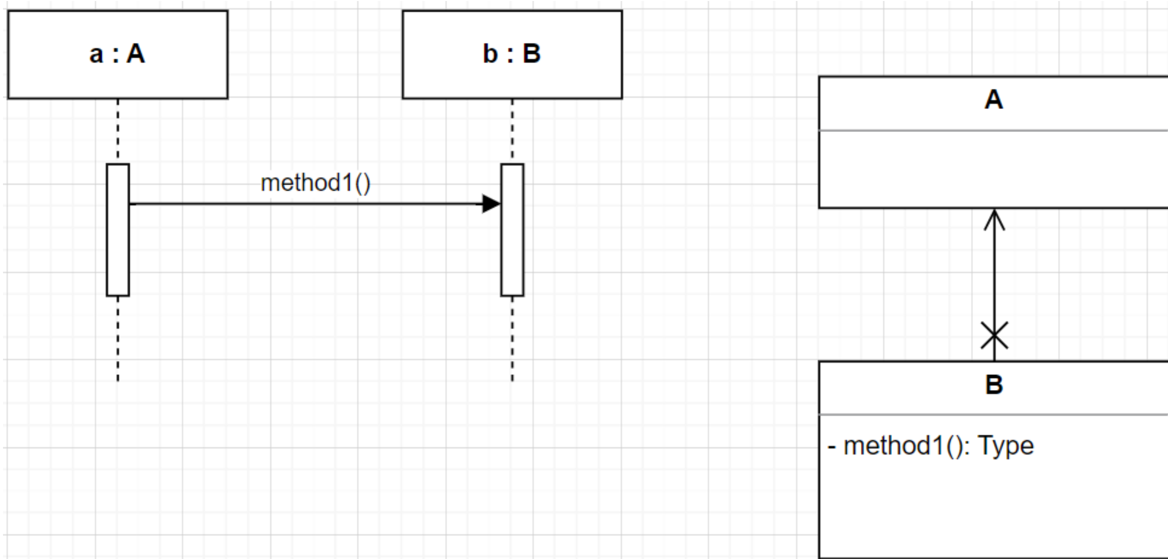

Рис.13. Приклад протиріччя: Звернення до класу з некерованим кінцем асоціації.

Протиріччя 11

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.

Приклад протиріччя зображений на рисунку 14.

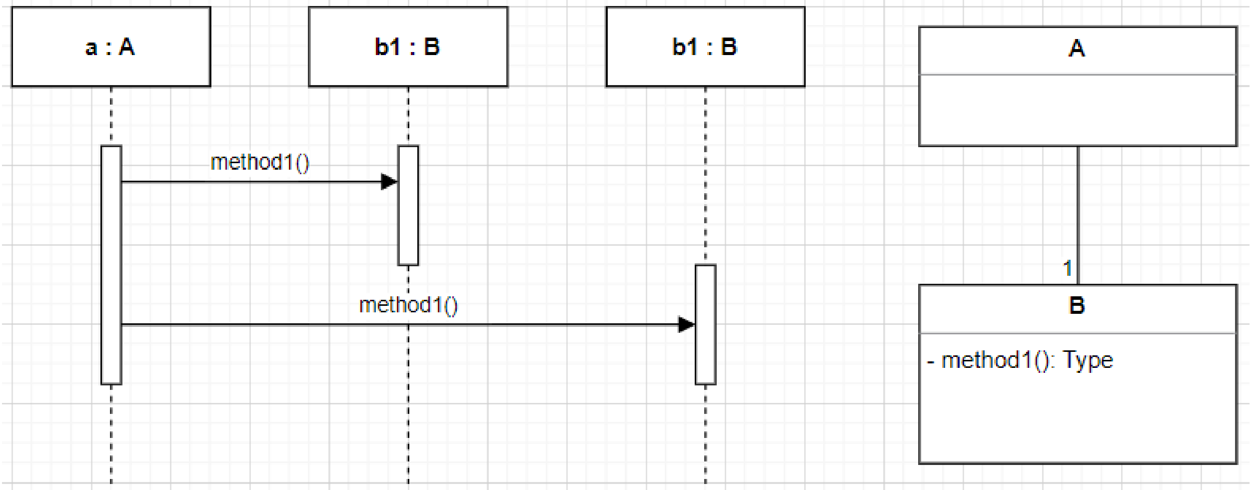

Рис.14. Приклад протиріччя: Звернення до двох об'єктів, коли регламентований тільки один.

Протиріччя 12

Об'єкти класу зі стереотипом «utility» не можуть існувати, бо такий клас не може існувати [6, с.682].

Приклад протиріччя зображений на рисунку 15.

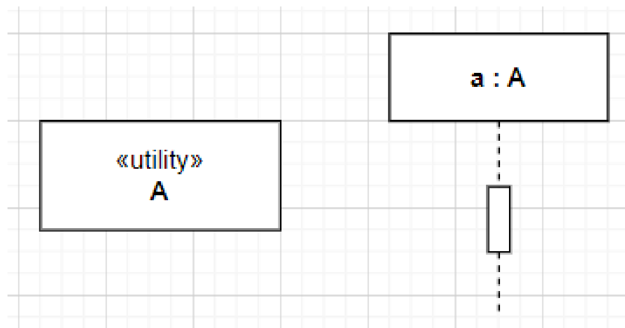

Рис.15. Приклад протиріччя: Ініціалізація класу зі стереотипом «utility».

2.1.3. 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 звязки між об’єктами не повинні суперечити тим, що вказані в діаграмі класів. Спираючись на це можна виділити такі протиріччя:

Протиріччя 13

Значення поля класу не сумісне з його типом, що вказаний у діаграмі класів.

Приклад протиріччя зображений на рисунку 16.

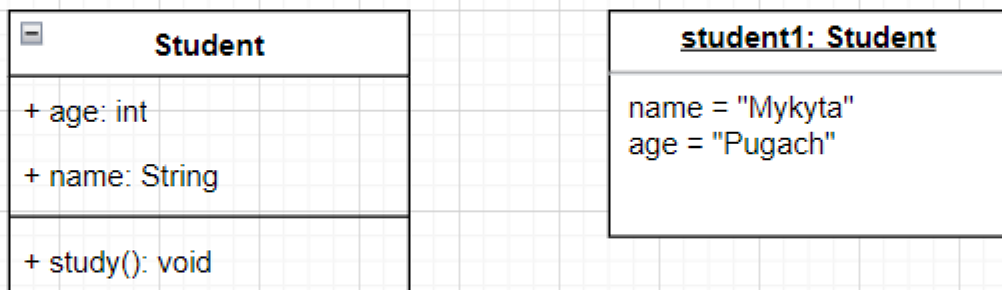

Рис.16. Приклад протиріччя: Значення поля несумісне з його типом.

Протиріччя 14

При композиції об’єкт частина може належати тільки одному об’єкту-цілому одночасно.

Приклад протиріччя зображений на рисунку 17.

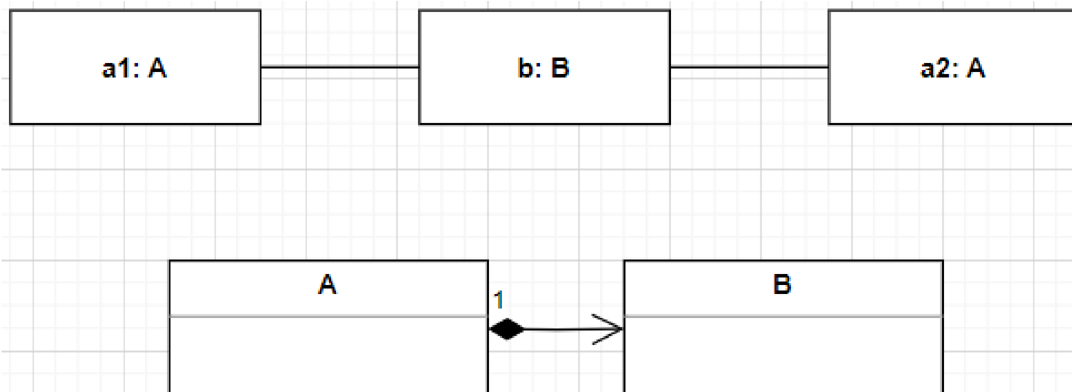

Рис.17. Приклад протиріччя: об’єкт - частина одразу двох цілих.

Протиріччя 15

При асоціації двох класів то кількість екземплярів першого класу та кількість екземплярів другого не повинні суперечити тому, яка кількість регламентована у діаграмі класів.

Приклад протиріччя зображено на рисунку 18.

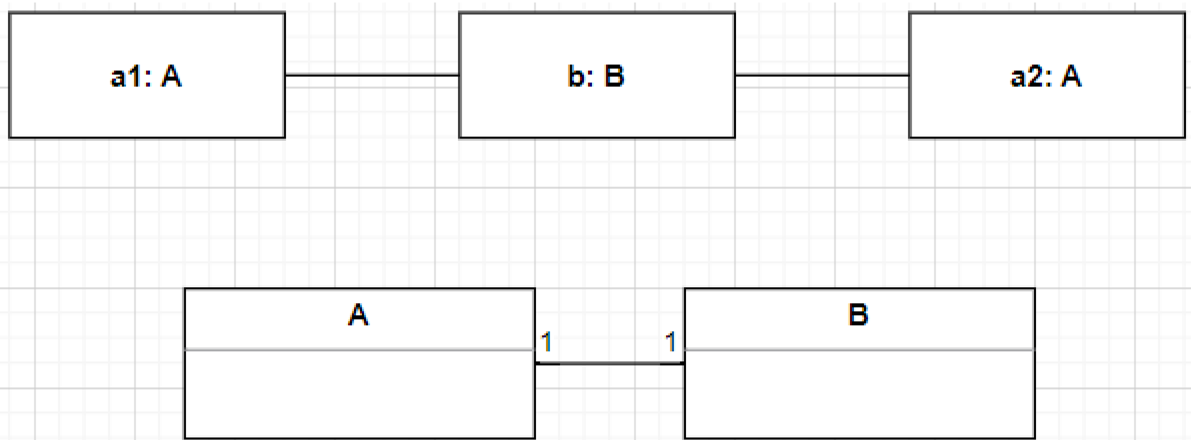

Рис.18. Приклад протиріччя: кількість екземплярів, що взаємодіють не відповідає кратності асоціації.

Протиріччя 16

Об'єкти класу зі стереотипом «utility» не можуть існувати, бо такі класи не можна ініціалізувати [6, с.682].

Приклад протиріччя зображено на рисунку 19.

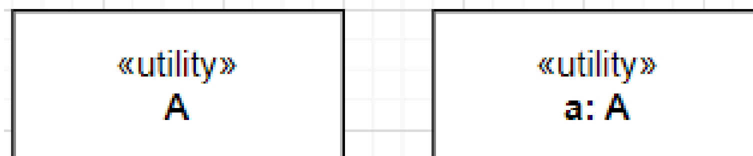

Рис.19. Приклад протиріччя: проініціалізовано клас зі стереотипом «utility».

Протиріччя 17

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.

Приклад протиріччя зображено на рисунку 20.

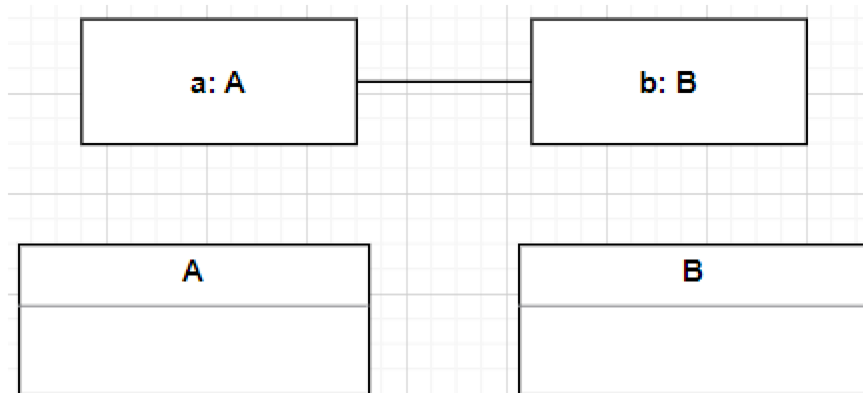

Рис.20. Приклад протиріччя: об'єкти пов'язані всупереч тому, що відповідні їм класи не мають зв'язків.

Протиріччя 17

Об'єкт класу, що має стереотип «implementationClass», не може бути екземпляром більш ніж одного класу. Те саме стосується і його нащадків [6, с.681].

Приклад протиріччя зображено на рисунку 21.

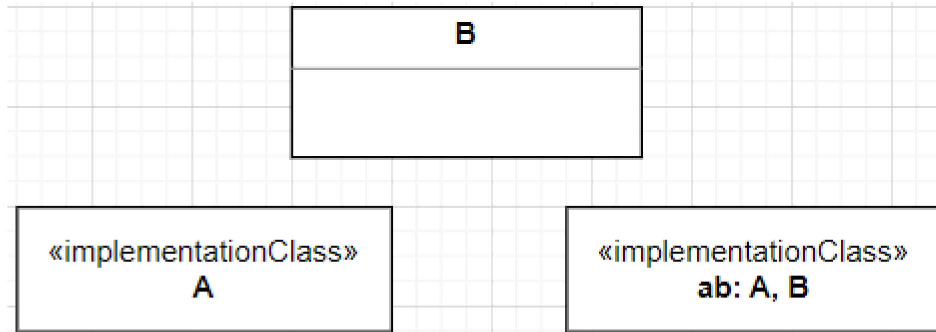

Рис.21. Приклад протиріччя: об'єкт класу зі стереотипом «implementationClass» є екземпляром більш ніж одного класу.

Протиріччя 18

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.

Приклад протиріччя зображено на рисунку 22.

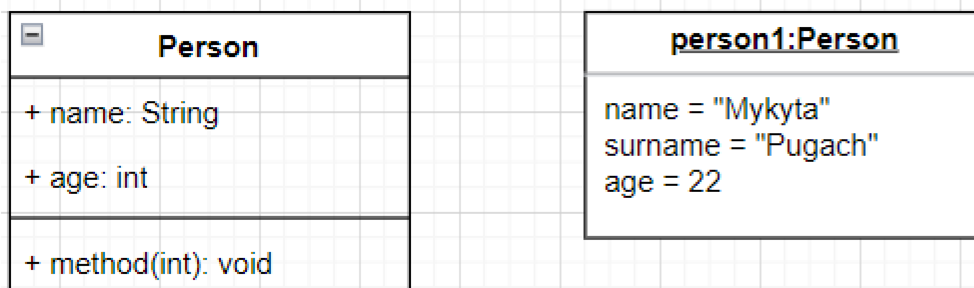

Рис.22. Приклад протиріччя: об'єкт має атрибут, який не вказано у відповідному класі.

2.1.4. 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.

Протиріччя 19

Послідовність відправлень повідомлень у діаграмі послідовностей не повинна суперечити множині послідовностей переходів у діаграмі станів даного класу.

Приклад протиріччя зображено на рисунку 23.

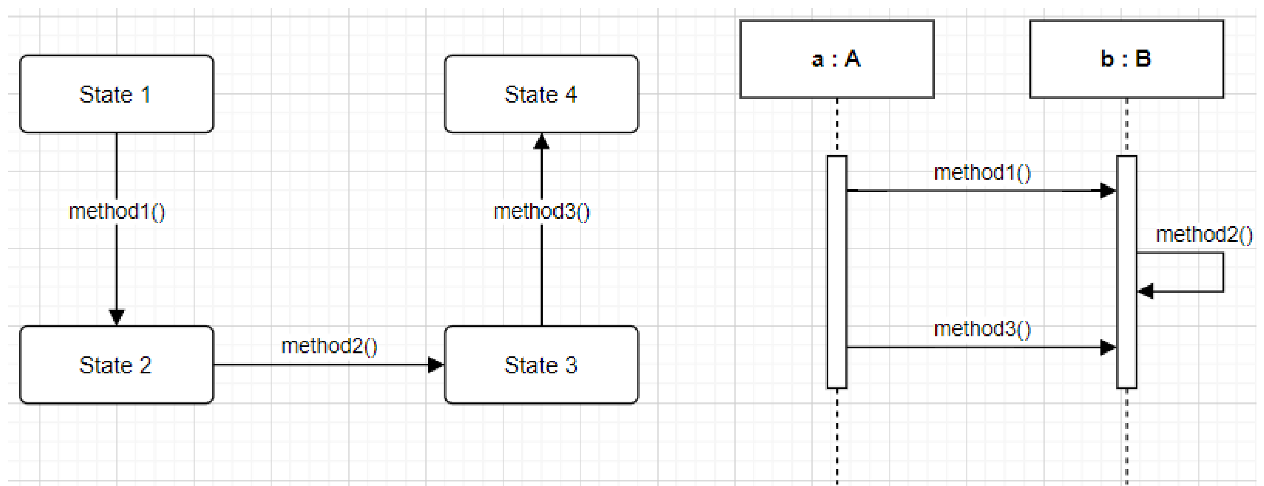

Рис.23. Приклад протиріччя: Послідовність відправлень повідомлень у діаграмі послідовностей суперечить послідовності переходів у діаграмі станів.

Протиріччя 20

Кожен метод, що використовується для зміни стану об’єкта у діаграмі станів, повинен бути описаним у діаграмі класів.

Приклад протиріччя зображено на рисунку 24.

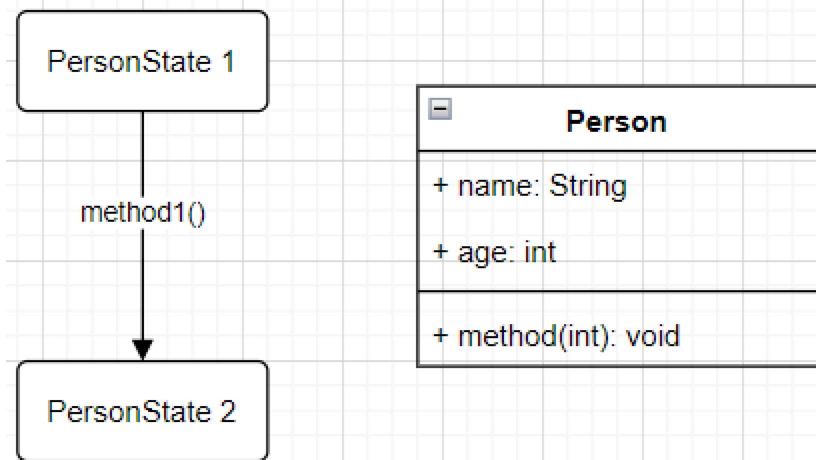

Рис.24. Приклад протиріччя: Використання у діаграмі станів методу, що не описаний у діаграмі класів.

2.1.5. Протиріччя всередині діаграми станів

Протиріччя 21

Кожен стан об’єкту повинен бути досяжним з початкового стану.

Приклад протиріччя зображено на рисунку 25.

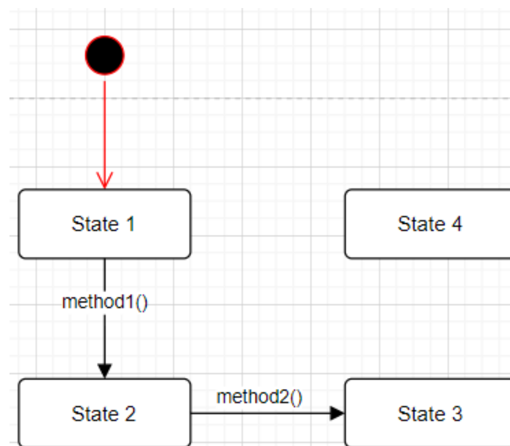

Рис.25. Приклад протиріччя: Недосяжний стан у діаграмі станів.

Протиріччя 22

Кінцевий стан завжди має бути досяжним.

Приклад протиріччя зображено на рисунку 26.

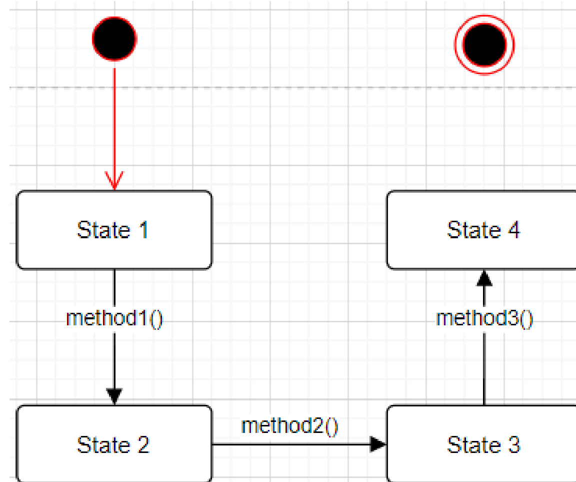

Рис.26. Приклад протиріччя: Кінцевий стан є недосяжним у діаграмі станів.

2.2. Опис застосунку для створення UML діаграм

Застосунок має назву «diagrams.net». Він надає широкі можливості по створенню різних типів діаграм, у тому числі UML. У ньому легко створювати діаграми класів, послідовностей та інші. Є можливість імпортувати діаграми, що були створені в інших сервісах. Також цей застосунок дає можливість імпортувати створену діаграму у вигляді XML файлу. Саме ця можливість стала основною при виборі застосунку.

Отримання XML файлу дає можливість створити програму, яка буде його зчитувати та шукати протиріччя.

Опис структури XML файлу:

Файл 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 в середині 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 усі 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

2.3. Опис методів пошуку протиріч

Головною метою роботи є написання програми, що буде розпізнавати протиріччя в межах однієї діаграми та між двома діаграмами. Так як ми можемо запрограмувати розпізнавання об’єктів діаграми з XML файлу, то ми можемо шляхом порівнянь і деяких правил запрограмувати пошук протиріч. Далі будуть описані алгоритми пошуку кожного з протиріч:

1. *У діаграмі послідовностей є клас, що не описаний у діаграмі класів*.

За результатами парсингу XML файлу, що може дати сервіс для побудови діаграм та класифікації елементів двох видів діаграм, програма має два списки класів: з діаграм класів та послідовностей. Беручи діаграму класів, як основну, ту що зберігає у собі усі можливі класи, що мають усі види діаграм, ми виділяємо ці класи, як основні, базові. Далі можна зіставити два списки класів. Ситуація, коли у списку з діаграми послідовностей не буде якогось класу з базового, цілком нормальна, бо не завжди будуть використовуватися всі класи

описаної програми. Але ситуація, коли в списку класів з діаграми послідовностей є клас, що не міститься у базовому списку, вийняткова. Тобто, порівнявши два списки, легко знайти описане протиріччя.

2. *Нелегітимне використання методів класу*.

Знову, як вхідні данні для пошуку протиріч маємо два списки класів. Як базовий – список утворений в результаті парсингу діаграми класів. Другий – утворений в результаті парсингу діаграми послідовностей. Під використанням методів мається на увазі виклики методів, відображенні у діаграмі послідовностей. Як результат парсингу та класифікації елементів діаграми послідовностей, ми маємо класи, що містять методи, які викликаються іншими класами. Ці методи зберігають у собі інформацію про класи, що їх викликають. Для того, щоб отримати інформацію про рівень доступу методу потрібно співставити класи з діаграми класів та з діаграми послідовностей. Методи з перших зберігають вичерпну інформацію про те, як можна метод використати. Ця інформація зберігається, як у моделі самих методів, наприклад, рівень доступу, список аргументів, значення, що повертається. Або в моделі класу – це залежності класу. В залежності від того, агрегація чи наслідування, protected метод не може або може бути використаним відповідно. Отже, можливість викликати метод іншого класу залежить від багатьох факторів, кожен із яких має бути перевіреним. На далі буде представлений опис перевірки кожного фактору:

2.1. *Перевірка чи є відповідний метод у класу, 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*. У одного класу може бути декілька методів з однаковою назвою, але з різним списком аргументів. Ця перевірка буде виконуватись першою і перевіряти чи є хоч один метод з такою назвою. Отже, для цього потрібно мати пару співставлених класів, тобто один і той самий клас, що описаний у двох різних діаграмах. Перевіряємо ми чи є метод описаний у діаграмі послідовностей. З класу з основної діаграми беремо список назв методів. Але в цьому списку не буде методів, що описані у батьківських класах. Отже стоїть задача отримати повний список доступних класу методів. Так як кожен клас, що утворився в результаті парсингу та класифікації діаграми класів, має список відносин. Відносини в свою чергу містять класи, що стоять у залежності та тип залежності. Тоді рекурсивно ми можемо отримати всі батьківські класи для даного. В результаті отримавши список усіх батьківських класів, з них потрібно взяти всі методи окрім, тих, що мають рівень доступу private. Отже тепер маємо список всіх доступних класу методів. Останнє – треба отримати список назв методів і перевірити чи є серед них потрібна.

2.1.2. *Список аргументів*. Наявність у класу методу з потрібним ім'ям не дає гарантії, що це саме той метод. Для пошуку даного протиріччя будуть використовуватися вже зроблені кроки з перевірки назви. В даному випадку також потрібно знайти усі доступні класу методи, а отже потрібен список всіх супер класів. З наявних у класі методів та усіх не private методів суперкласів утворюємо один список. Далі потрібно серед них залишити тільки ті, що мають однакове з порівнюваним ім'я. Далі серед них потрібно знайти метод з однаковим списком аргументів.

2.1.3. *Значення, що повертається*. Останнє, чим можуть відрізнятись методи. Тепер мова буде йти про порівняння методів, що мають однакову назву та список аргументів. Саме такий метод буде знайдено після виконання перевірки на список аргументів. Отже отримавши з класу, що наявний у діаграмі класів метод, просто порівнюємо значення, що повертається.

2.2. *Перевірка на легітимність виклику методу*.

У кожного методу є свій рівень доступу. І в залежності від того це private, protected, public або default, той або інший клас може або не може викликати метод. Отже, на вхід отримуємо пару співставлених класів, знову – з діаграми класів та послідовностей. Ця перевірка буде проводитись після того, коли ми впевнились, що метод, який викликається в діаграмі послідовностей існує в діаграмі класів. Тож тепер потрібно знайти відповідні один одному методи в двох класах. Наступним кроком буде заповнити у моделі методу поле «рівень доступу». Взагалі, коли у діаграмі послідовностей позначається виклик методу - це означає, що цей метод має відповідний рівень доступу. Але, по-перше, в процесі класифікації елементів діаграми послідовностей не можливо напевно його визначити, а, по-друге, може бути помилка, котру, власне, описана програма має знайти. Отже, з відповідного класу переносимо значення рівня доступу і далі будемо перевіряти чи відповідає, описана у діаграмі послідовностей, схема його викликів цьому рівню.

По-перше, опишемо правила, за якими клас може викликати метод іншого класу для кожного рівня доступу:

2.2.1. *Public*. Найбільш відкритий рівень доступу. Єдине обмеження, яке накладається на виклик методу – це наявність залежності між класом, що викликає метод та класом, що його містить. Отже, ми

маємо пару співставлених класів. В результаті класифікації елементів діаграми послідовностей клас, що там описаний не матиме списку залежностей, але його методи матимуть опис класу, що їх викликає. А клас, отриманий в результаті класифікації елементів діаграми класів матиме список залежностей. Отже треба взяти клас, що позначений у методі, як той що його викликає та спробувати знайти його серед списку залежностей. Якщо його там нема – тоді виклик методу неможливий і це є протиріччям двох діаграм.

2.2.2. *Private*. Навпаки – найбільш закритий рівень доступу. Доступ мають лише об'єкти цього самого класу. Отже і викликати його може або об'єкт сам у себе, або в іншого об'єкту того самого класу. Для того, аби це перевірити знову потрібно взяти клас, що викликає метод і порівняти з тим класом, що цей метод містить. Виклик буде вважатись обґрунтованим тільки якщо вони співпадають, інакше – протиріччя.

2.2.3. *Protected*. Особливістю цього рівня доступу є те, що його можуть викликати тільки сам цей клас або його нащадком. Отже, для пошуку цього протиріччя стоїть задача знайти всіх нащадків класу. Для цього візьмемо два класи: той, що містить метод, і той, що його викликає. При цьому клас, що викликає буде вважатись потенційним нащадком. Далі зведемо цю задачу до подібної тій, що була вирішена для попередніх протиріч, а саме – знайти всі батьківські класи для 'потенційного нащадка'. А далі перевірити чи є серед них потрібний. В результаті ми перевіримо чи є він нащадком, але protected метод, звісно, може викликати клас і сам у себе. Тобто, треба зробити ту саму перевірку, як і для private.

2.2.4. *Default або Package Private*. Як говорить друга назва цього рівня доступу, він інкапсулює інформацію в межах одного пакету класів. Отже, для перевірки легітимності виклику методу, знову візьмемо клас, що викликає з моделі методу і клас, що цей метод містить. Далі перевіримо чи в одному пакеті знаходяться ці класи, якщо ні, то доступ у першого до методу заборонений, а отже відображення його виклику на діаграмі послідовностей – це протиріччя з діаграмою класів.

3. *Приналежність об'єкта-частини до кількох об'єктів-цілих при композиції.*

Це протиріччя виникає всередині діаграми класів. Композиція говорить про те, що якийсь об'єкт буде будуватись на основі інших об'єктів. При цьому не можна, щоб один об'єкт був частиною для кількох інших об'єктів. Це протиріччя можна виявити в двох ситуаціях: коли кратність композиції більше 1, або коли об'єкт є частиною більш ніж однієї композиції. Отже перевірити потрібно обидва варіанти.

3.1. *Кратність композиції більше 1.* На вхід беремо список всіх класів з діаграми класів. З кожного класу беремо список усіх його залежностей. Далі об'єднуємо всі ці списки в один. Так як протиріччя стосується виключно композиції, потрібно вибрати зі списку всіх залежностей діаграми усі залежності, у яких тип «Композиція». У кожної залежності композиції є поле інформації, в даному випадку це кратність. Отже тепер потрібно вирахувати чи є серед цих композицій така, у якої кратність більше 1. Для цього з рядка кратності видалимо усі символи, що будуть валідними для композиції з кратністю 1 або менше. Перерахуємо всі варіанти: 0, 1, 0..1. Далі використаємо регулярний вираз для пошуку цих варіантів: «[0-1\.\s]». Отже, якщо композиція має

кратність більше одиниці, то після видалення з рядка кратності всіх символів, що підходять під регулярний вираз повинні залишитись ще хоча б один символ. Це може бути n або число більше за 1, наприклад. Отже треба знайти усі такі композиції, де кратність більше одного. А клас, що є об'єктом-частиною з протиріччям буде вказаний в полі «пов'язаний клас».

3.2. *Об'єкт є частиною більш ніж однієї композиції.* На вхід знову маємо список усіх класів з діаграми класів. Як і у минулому випадку, потрібно отримати всі залежності з діаграми, що є композиціями. Збережемо всі композиції у список. Далі утворимо ще один список, що буде містити класи, що описані як залежні в композиціях. Отже, тепер протиріччя буде якщо у списку цих класів деякі будуть повторюватись. Це означає, що якісь класи є чистинами в декількох композиціях. Перевірити це можна так: з композицій знов утворимо новий список класів, що є там частинами. Потім використаємо метод distinct(), він поверне список тільки унікальних класів. Далі використовуємо метод Collections.frequency(), першим аргументом буде список всіх класів, з можливими повторюваннями, а другий це клас, із списку унікальних. Цей метод поверне кількість входжень у колекцію, що була першим аргументом. Отже далі знаходимо всі класи, що входять більше одного разу, це і буде знайдене протиріччя.

4. *Циклічна залежність класів.*

Це протиріччя також виникає всередині діаграми класів. Циклічна залежність виникає, коли клас має залежність із самим собою через ланцюжок залежностей з іншими класами. Дане протиріччя сигналізує про погано спроектований дизайн. Для пошуку даного протиріччя дуже зручно уявити, що

діаграма класів – це орієнтований граф, у якому вершинами є класи, а ребра – це залежності. Далі треба шукати вже не цикли у діаграмі, а цикли у графі. Це доволі поширена задача і вирішується вона з використанням пошуку в глибину. Тобто тепер для знаходження циклів у діаграмі треба зробити два кроки: по-перше, треба записати усі класи та залежності у виді зручному для роботи, як з графом, по-друге, реалізувати пошук в глибину.

5. *Несумісні стереотипи класу.*

Це протиріччя виникає в середині діаграми класів. Опис класу в цій діаграмі може мати один, або декілька стереотипів, однак не всі стереотипи можуть одночасно бути в одного класу. Наприклад, клас не може одночасно бути enumeration та interface. Після парсингу та класифікації діаграми класів, інформація про стереотипи класу знаходяться у полі назви. Це протиріччя можна виявити перевіривши усі класи, що мають більше одного стереотипу. Для цього треба виписати усі стереотипи, що не мають одночасно належати одному класу, і перевірити чи є такі комбінації в цих класах.

2.4. Опис застосунку 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.

На рисунку 27 буде наведено загальну схему роботи застосунку.

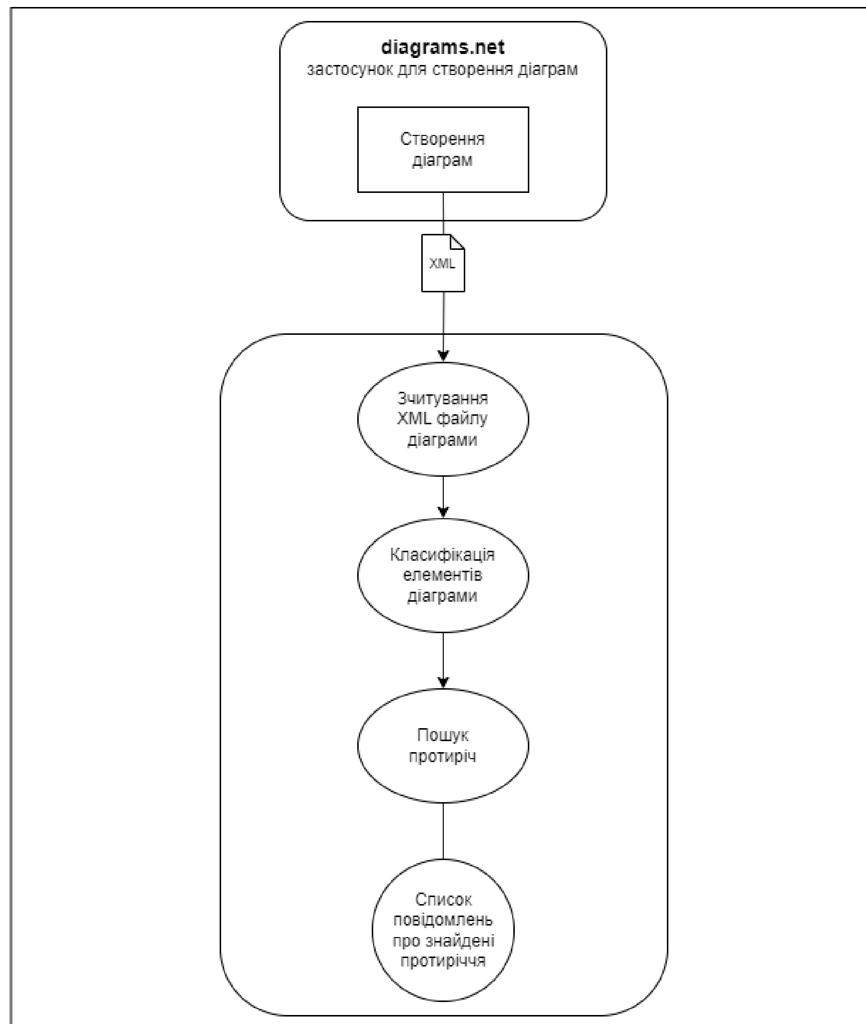

Рис.27. Загальна схема роботи застосунку.

Для вирішення проблеми пошуку протиріччя роботу програми можна розбити на два етапи:

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

Далі розберемо як саме програма реалізує ці етапи:

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

Перед використанням програми користувач має діаграму представлену у виді XML файлу. Для того що програма могла обробити цей файл, в ній потрібно створити сутності, що будуть відповідати елементам діаграми, наприклад, клас або метод. Також, так як XML файл використовує інші сутності для зберігання діаграми, то потрібно у програмі створювати і їх. Тож далі буде представлений опис двох типів сутностей: сутності XML файлу та сутності UML діаграм класів та послідовностей.

1.1. *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.***

Діаграма класів пакету ***models.xml*** зображено на рисунку 28:

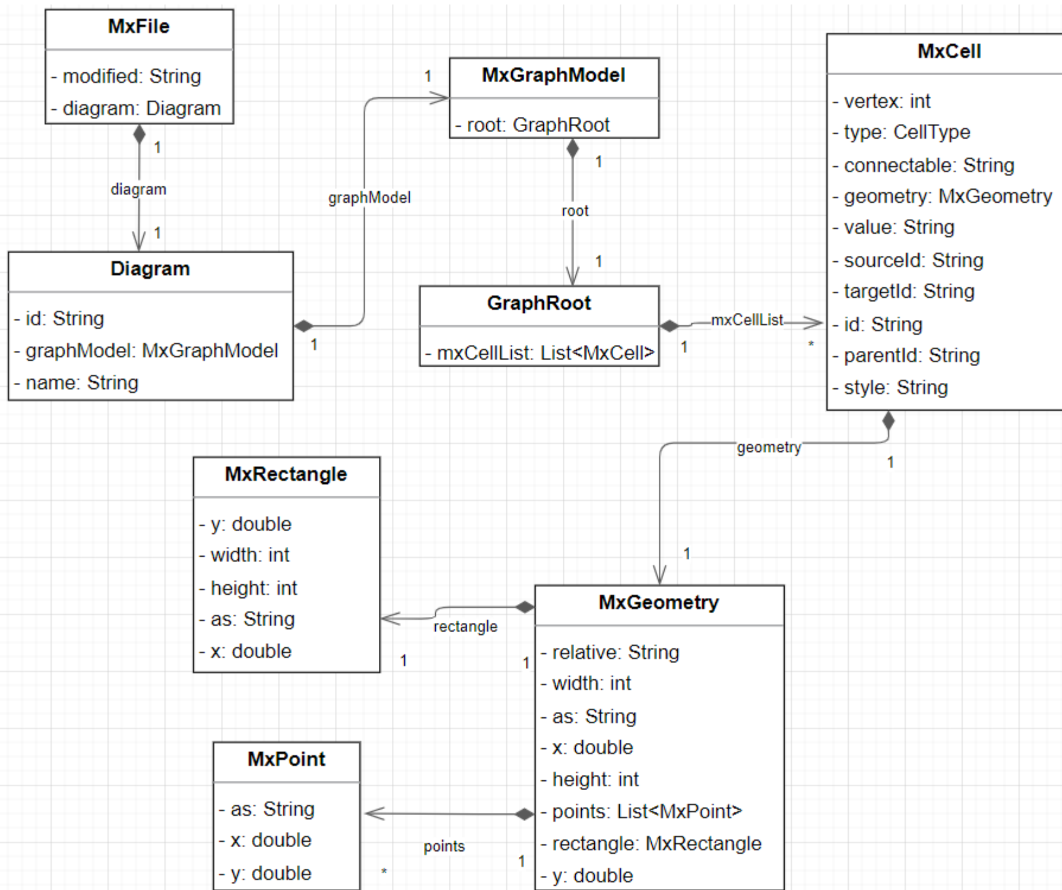

*Рис.28. Діаграма класів пакету **models.xml**.*

Кожен клас відображає вміст XML тегу з тією самою назвою.

1.2. *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 повина зберігати діаграму саме в таких сутностях. Для цього були створені спеціальні класи, що зберігаються у пакеті ***models.design***.

Діаграма класів пакету ***models.design*** зображено на рисунку 29:

Рис.29. Діаграма класів пакету ***models.design****.*

В цілому сенс всіх класів досить зрозумілий з їх назв. Класи містять методи та поля, мають зв'язки з іншими класами. Окремих пояснень потребують два класи: *DiagramArrow* та *SequenceDiagramMethod.* Взагалі, зрозуміло, що DiagramArrow позначає стрілку, що є майже у всіх UML діаграмах. Але по суті стрілка говорить про взаємозв'язок класів, або про відправку повідомлення. Це є сутніть, що відображає стрілка, і ця сутність міститься у класі DiagramRelation. Клас DiagramArrow створений, як проміжна сутність між

тегами XML і елементами діаграм, бо в XML файлі інформація про стрілку міститься у декількох тегах, і зручно перед створенням класу залежності зібрати всю інформацію в одному місті. Що стосується SequenceDiagramMethod, то це метод, що був класифікований з діаграми залежностей. Так як діаграма залежностей показує які класи або об'єкти викликають методи, що є дуже корисною інформацією для пошуку протиріч, то програма зберігає цю інформацію.

1.3. *Зчитування та парсинг XML файлу.*

Тепер, коли програма має 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***.

Діаграма класів пакету ***parser*** зображено на рисунку 30:

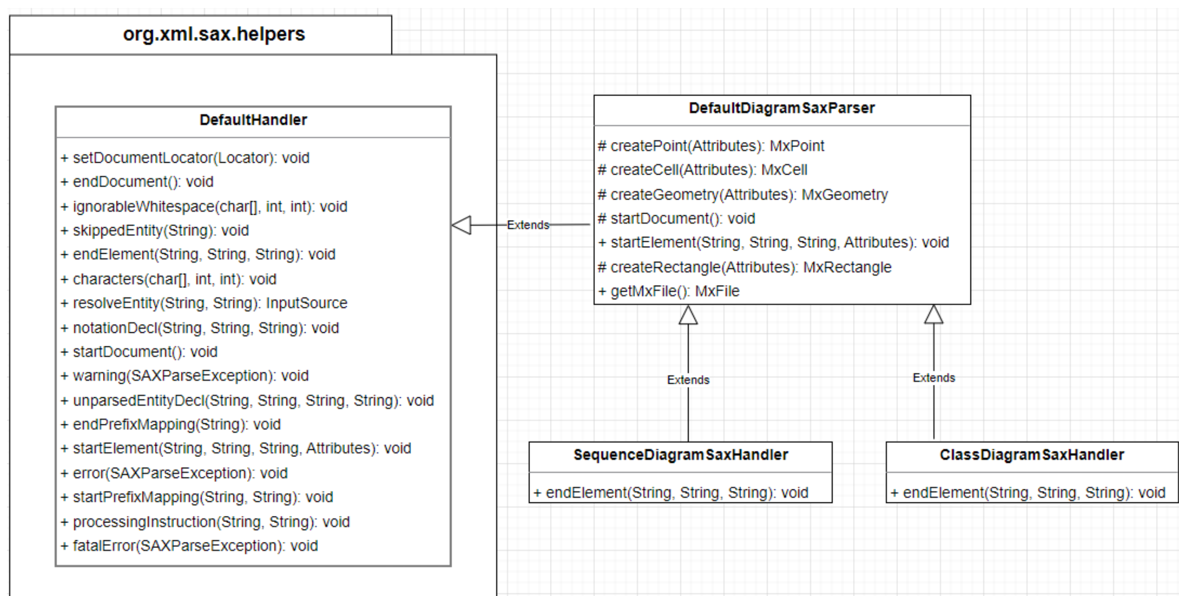

Рис.30. Діаграма класів пакету ***parser****.*

Перше, що видно з діаграми, що для того аби розпарсити XML було успадковано клас *DefaultHandler* з пакету ***org.xml.sax.helpers***. Три перевизначенні методи: *startDocument()*, *startElement()* і *endElement()* дозволяють досить гнучко заповнювати інформацію в об'єкти класів з пакету ***models.xml***. Друге – це різна реалізація парсингу для діаграми класів та діаграми послідовностей. Корінь цього рішення полягає у тому, що зовнішній вигляд одних і тих самих сутностей у діаграмах класів та послідовностей може сильно відрізнятись. Наприклад, клас у діаграмі класів – це просто прямокутник, а в діаграмі послідовностей клас, або об'єкт класу ще має пунктирну лінію вниз та часто характерний видовжений прямокутник на ній. Отже, зважаючи на те, що XML файл описує саме зовнішній вигляд діаграми, треба по-різному парсити ці два види діаграм.

1.4. *Класифікація елементів UML діаграм.*

Наступна задача з набору об'єктів класів з пакету ***models.xml***, що були отриманні 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***.

Діаграма класів пакету ***classification*** зображено на рисунку 31:

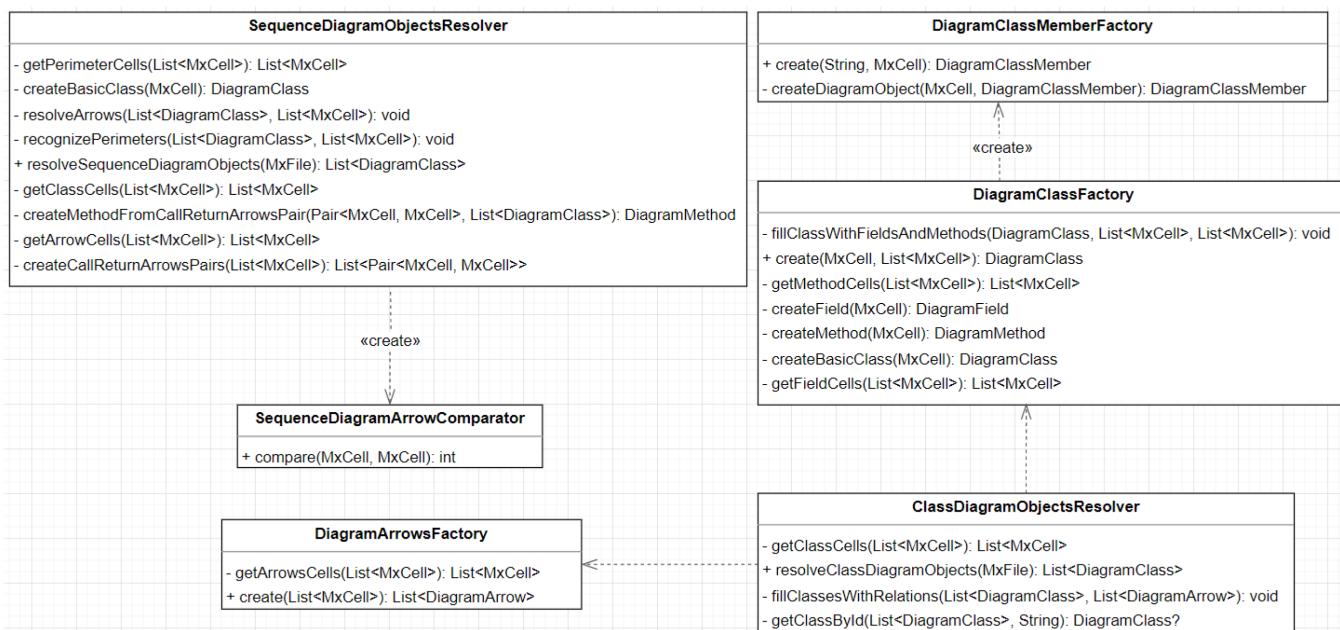

Рис.31. Діаграма класів пакету ***classification***.

Отже, знову можна побачити, що діаграми класів та послідовностей обробляються різними класами. Насправді, хоча ці дві діаграми описують ті самі сутності, але у їх візуальному відображені дуже багато відмінностей. Тому алгоритми класифікації не схожі і не мають загального коду. Детальний опис роботи методів цих класів був у розділі «Опис відповідностей між тегами xml та елементами діаграми»**.** Ці методи вирішують рівно ті задачі та використовують ті особливості, що були там описані. В результаті роботи резолверів програма має список об'єктів класу *DiagramClass*, що в свою чергу зберігає в собі об'єкти інших класів пакету ***models.design***.

1.5. *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***.

Діаграма класів пакету ***conflicts*** зображено на рисунку 32:

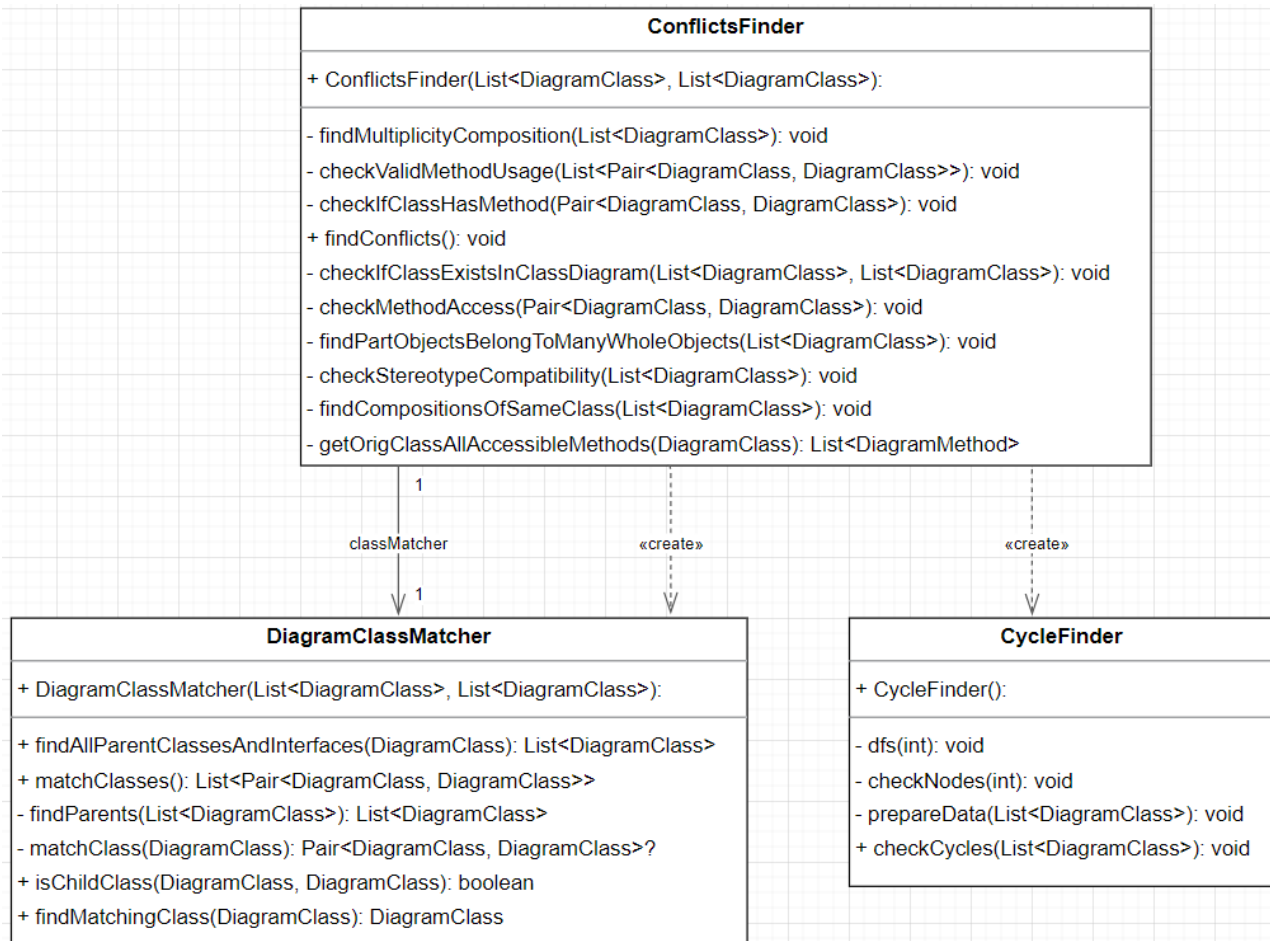

*Рис.32. Діаграма класів пакету **conflicts**.*

Основний клас *ConflictsFinder* для пошуку на вхід отримує два списки класів. Перший – отриманний після класифікації діаграми класів, другий – діаграми послідовностей. Так як обидві діаграми описують одні й ті самі сутності, для пошуку протиріч дуже зручно їх зіставити. Цим займається клас *DiagramClassMatcher*. Останній клас: *CycleFinder* вирішує задачу пошуку циклів у діаграмі класів. Описи та алгоритми пошуку всіх протиріч є у розділі «Пошук протиріч».

Тестування додатка

Далі будемо розглядати конкретний приклад двох діаграм: класів та послідовностей і те, які протиріччя будуть знайдені у результаті роботи програми.

Отже, розглянемо діаграму класів. Вона зображена на рисунку 33.

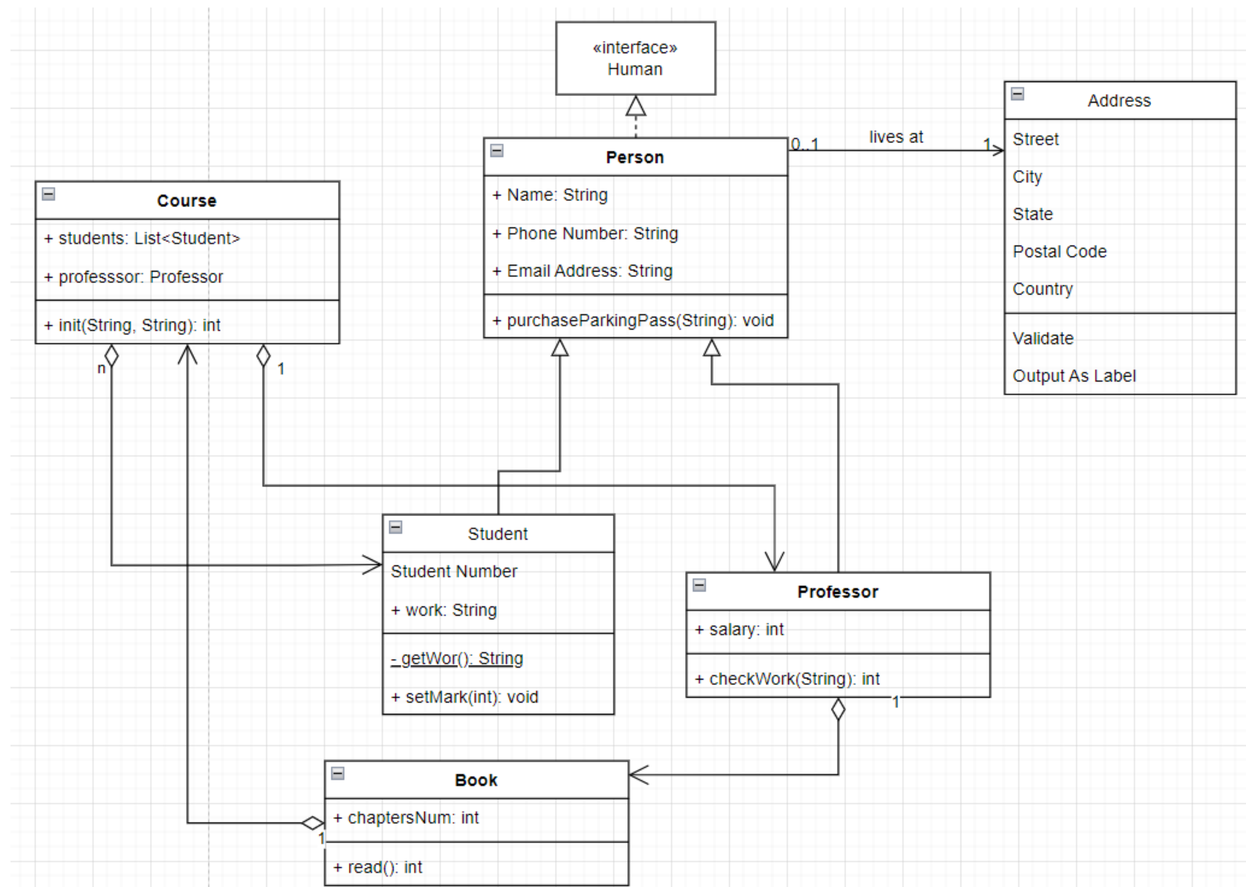

Рис.33. Діаграма класів для тестування додатка.

Маємо шість класів, інтерфейс, багато залежностей між цими класами. За цією діаграмою класів побудовано наступну – діаграму послідовностей. Вона зображена на рисунку 34.

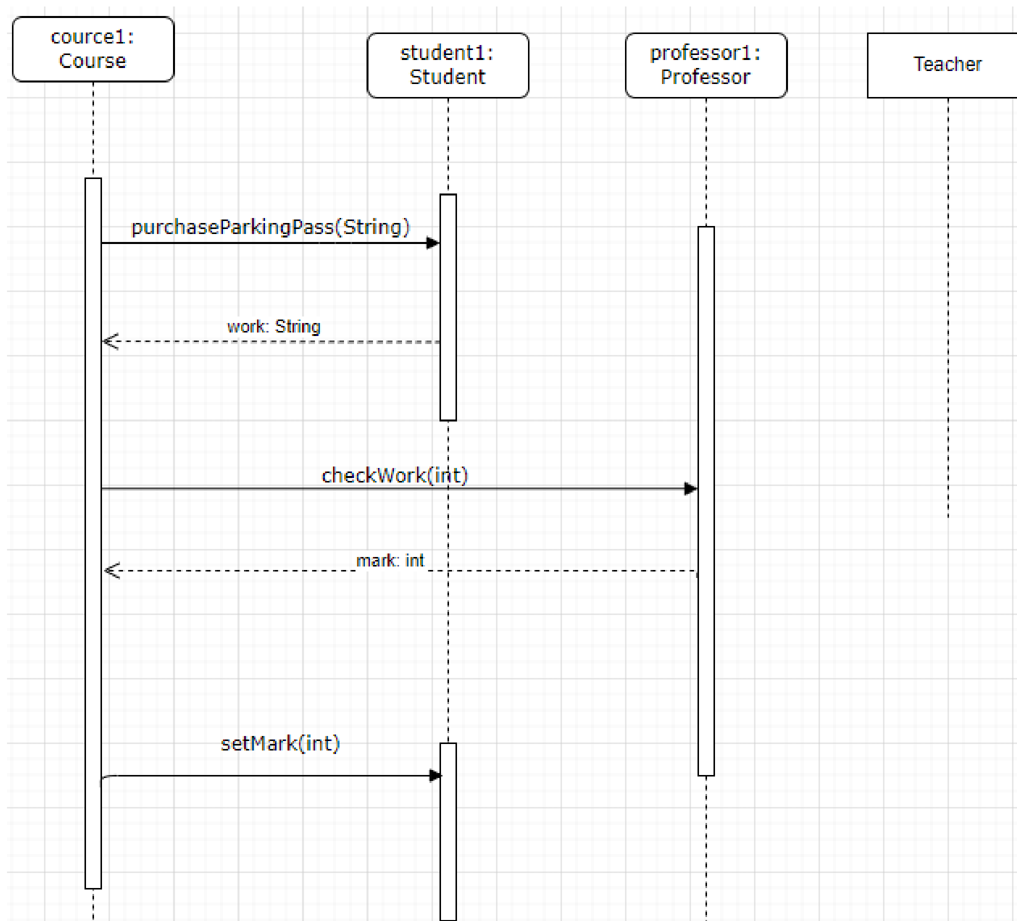

Рис.34. Діаграма послідовностей для тестування додатка.

На цій діаграмі можна побачити як взаємодіють класи. Але в цих двох діаграмах навмисно закладені протиріччя. І далі розглянемо результат роботи додатка. Додаток виявив такий список протиріч:

1. Діаграма класів має цикли
2. На діаграмі класу немає класу «Teacher»
3. Немає такого методу «purchaseParkingPass» із такими аргументами чи типом!
4. Немає такого методу «checkWork» з такими аргументами або типом!

Отже, бачимо, що було знайдено 4 протиріччя. Перший був виявлений лише у діаграмі класів, вона дійсно має цикли. Інші були знайдені на перетині двох діаграм: класів та послідовностей. Класу під назвою «Teacher» дійсно не описано у діаграмі класів. Метода «purchaseParkingPass» не існує у класу «Student». А у методу «checkWork» насправді інший список аргументів.

Отже, окрім цього прикладу, програма була багаторазово протестована на багатьох різних прикладах і було доведено, що вона коректно знаходить протиріччя у діаграмах об’єктно орієнтованого дизайну.

ВИСНОВКИ

У результаті проведеного дослідження було досягнуто основної мети роботи. Було проаналізовано великий обсяг літератури та специфікацій з об’єктно орієнтованого дизайну, в результаті чого, було виявлено близько двадцяти протиріч. Були розроблені методи та алгоритми знаходження та виявлення протиріч. А також реалізовано додаток на мові програмування Java, який успішно реалізує виведені протиріччя.

Отримані результати підкреслюють важливість вивчення та врахування протиріч на етапі проектування програмного забезпечення. Недоліки та розроблені методи, можуть допомогти суттєво покращити процес проектування та забезпечити більшу ефективність та якість програмного забезпечення, що розробляється. А програмний продукт може дати поштовх до покращення розробок у даній області.

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.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ajit Singh, Ms. Anamika. Object Oriented Modeling and Design Using UML: 2nd Edition, 2022. – 153 p.
2. Stephen J. Mellor, Marc J. Balcer. Executable UML: A Foundation for Model-Driven Architecture, 2022. – 402 p.
3. Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, and Jim Conallen. Object-Oriented Analysis and Design with Applications 3rd Edition, 2007. – 720 p.
4. Craig Larman. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 2004. – 736 p.
5. Vanessa Weber, Kleinner Farias, Lucian Gonçales, Vinícius Bischoff. Detecting Inconsistencies in Multi-view UML Models. International Journal of Computer Science and Software Engineering (IJCSSE), Volume 5, Issue 12, December 2016.
6. OMG. Unified Modeling Language 2.5.1 Specification. 2017.
7. Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John. Design Patterns: Elements of Reusable Object-Oriented Software, 1994. – 416 p.
8. Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship, 2008. – 264 p.
9. Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. Refactoring: Improving the Design of Existing Code, 1999. – 432 p.
10. Leszek A. Maciaszek. Requirements Analysis and System Design: Developing Information Systems with UML, 2001. – 378 p.
11. Shivam Singh. Introduction to System Design: Crack System Design Interviews and Understand how your favorite Tech products work, 2021. – 120 p.

12. Joyce Farrell. Programming Logic and Design, Introductory, 2017. – 384 p.
13. Cross-Diagram UML Design Verification. V. Ermolayev et. al. (eds.) ICT in Education, Research and Industrial Applications. CCIS, Vol. 347, Springer-Verlag, Berlin Heidelberg (2013). – pp. 165-176. Oleksandra Kulankhina, Hlib Mykhailenko.
14. 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. DOI: 10.5815/ijitcs.2018.09.06 pp.47-56.
15. Rational Rose. https://www.ibm.com/docs/en/rational-clearquest/7.1.0?topic=developing-schemas-clearquest-designer
16. Diagrams.Net. https://app.diagrams.net/

АНОТАЦІЯ

Дана дипломна робота складається зі вступу, основної частини, висновка та списку використаних джерел. Обсяг роботи становить 49 сторінок. Основна частина займає 39 сторінок та містить 33 рисунки. Документ містить 16 посилань на використані джерела.

Дане дослідження присвячене проблемі виявлення та пошуку протиріч у об'єктно орієнтованому дизайні. Метою цього дослідження є розробка методів та алгоритмів для виявлення та пошуку протиріч в об'єктно орієнтованому дизайні з метою покращення якості проектування, а також написання програмного забезпечення, що буде реалізовувати дані алгоритми та методи. Для досягнення мети було досліджено потенційні недоліки та протиріччя, що можуть виникати під час проектування програмних систем.

Результатом дослідження є формулювання близько двадцяти протиріч на перетині чотирьох різновидів UML діаграм, а саме діаграми: класів, послідовностей, станів та об'єктів. Були розроблені методи та алгоритми виявлення та пошуку протиріч, а також реалізовано програмне забезпечення, що реалізує ці методи та алгоритми, та ефективно знаходить протиріччя у UML діаграмах об'єктно орієнтованого дизайну.

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

ANNOTATION

This diploma thesis consists of an introduction, the main part, a conclusion and a list of used sources. The volume of work is 49 pages. The main part occupies 39 pages and contains 33 figures. The document contains 16 references to used sources.

This study is devoted to the problem of identifying and finding inconsistency in object-oriented design. The purpose of this research is to develop methods and algorithms for detecting and searching for contradictions in object-oriented design in order to improve the quality of design. And also to write software that will implement these algorithms and methods. To achieve the goal, potential contradictions that may arise during the design of software systems were investigated.

The result of the study is the formulation of about twenty contradictions at the intersection of four types of UML diagrams: class, sequence, state and object. Methods and algorithms for detecting and finding contradictions have been developed, and software ,that implements these methods and algorithms and effectively finds contradictions in UML diagrams of object-oriented design.

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