КВАЛІФІКАЦІЙНА РОБОТА

Група МІПЗс-22

Борщ В.В.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Борщ Володимир Володимирович

УДК 004.4

Підвищення ефективності та оптимальності моделей та методів процесів дебагінгу

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль

_____Стисло О.В.

(підпис, дата, розшифрування підпису)

Студент

_____ Борщ В.В.

(підпис, дата, розшифрування підпису)

Допускається до захисту

Завідувач кафедри

_____к.т.н., доц. Ващишак С.П.

(підпис, дата, розшифрування підпису)

Керівник роботи

_____ к.т.н., доц. Демчина М.М.

(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «магістр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

« 19 » лютого 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Борщ Володимир Володимирович

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Підвищення ефективності та оптимальності моделей та методів процесів дебагінгу

керівник роботи:

Демчина Микола Миколайович, кандидат технічних наук, доцент

затверджена наказом вищого навчального закладу від « 26 » червня 2023 року № 32/1 с

2. Термін подання студентом роботи 16.02.2024

3. Вихідні дані роботи: Формальні моделі та методи процесу дебаггінгу

4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)

   1. Аналіз інструментів інтерпретації помилок під час процесу дебаггінгу
   2. Дослідження оптимальних моделей та структури представлення програмних помилок
   3. Методика інтерпретації та розуміння помилки користувачем
   4. Методологія візуалізації повідомлень про помилки компілятора

5. Дата видачі завдання 29.06.2023

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Розділ	Консультант (прізвище, ініціали та посада)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Аналіз інструментів інтерпретації помилок під час процесу дебаггінгу	26.09.2023	Виконано
2.	Дослідження програмних інструментів аналізу та інтерпретації помилок	20.10.2023	Виконано
3.	Дослідження методології розробки повідомлень про помилки	15.11.2023	Виконано
4.	Розробка методів та методології підвищення ефективності інтерпретації помилок	30.11.2023	Виконано
5.	Формування висновків	09.12.2023	Виконано
6.	Оформлення пояснювальної записки	22.12.2023	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	11.01.2024	Виконано

Студент _____ Борщ В.В.

(підпис) (прізвище та ініціали)

Керівник роботи _____ Демчина М.М.

(підпис) (прізвище та ініціали)

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу
23	Представлення помилки «неочікуваної конкатенації рядків» в IDE Visual Studio Code	64	Вигляд батьківських класів
27	Теоретичний фреймворк раціональної реконструкції	65	Завдання помилок компілятора
47	Типова модель конвеєра перевірки	68	Приклади подання помилок
52	Представлення помилок в сучасних IDE	70	Частота візуальних анотацій у Pilot

АНОТАЦІЯ

Кваліфікаційна робота присвячена підвищенню ефективності та оптимальності моделей та методів процесів дебаггінгу шляхом розробки концепції розширеного виведення пояснень про помилки компіляції.

В першому розділі проаналізовано основні труднощі які виникають з інтерпретацією повідомлень про помилки, створені інструментами програмного аналізу, що відповідно є значною причиною нездатності розробників усунути дефекти: труднощі з інтерпретацією повідомлень про помилки можна пояснити тим, що повідомлення про помилки визначають як недостатню раціональну реконструкцію як у візуальних, так і в текстових представленнях

В другому розділі проведено огляд інструментів аналізу програм з метою ознайомлення з поточними питаннями про те, як сучасні інструменти аналізу програм створюють і передають повідомлення про помилки розробникам. Повідомлення про помилки розглядаються через теоретичну структуру раціональної реконструкції, наголошуючи виборі дизайну, також помилки представлено в контексті інструментів аналізу програм, які розробники фактично використовують на практиці

В третьому розділі представлено дослідження повідомлень про помилки, використовуючи оптимальну модель аргументації Туліна та форму раціональної реконструкції пояснення. Було досліджено повідомлення про помилки та виконано їх класифікацію як правильні або неякісні структури аргументів.

КЛЮЧОВІ СЛОВА: ІНТЕРПРЕТАЦІЯ ПОМИЛОК, КОМПІЛЯТОР, ВІЗУАЛІЗАЦІЯ, ПОМИЛКИ РОЗРОБНИКА, ПРОГРАМНИЙ АНАЛІЗ, ПРОЦЕС ДЕБАГГІНГУ, ПОВІДОМЛЕННЯ ПРО ПОМИЛКУ

SUMMARY

The qualification work is devoted to increasing the efficiency and optimality of models and methods of debugging processes by developing the concept of extended derivation of explanations about compilation errors.

The first section analyzes the main difficulties that arise from the interpretation of error messages generated by software analysis, which, accordingly, is a significant reason for the inability of developers to eliminate defects: the difficulty with the interpretation of tool error messages can be explained by the fact that the error message is presented as an insufficient rational reconstruction as in both visual and textual representations

In the second section, an overview of application analysis tools is given, with an order to become familiar with current issues about how modern application analysis tools create and communicate error messages to developers. Error messages are examined through the theoretical framework of rational reconstruction, emphasizing sampling, and errors are presented in the context of program analysis tools that developers have actually made in practice.

The third section presents a study of error reporting using Thulin's optimal reasoning model and rational explanation reconstruction formulation. Error reports were examined and classified as valid or invalid argument structures.

KEY WORDS: ERROR INTERPRETATION, COMPILER, VISUALIZATION, DEVELOPER ERRORS, SOFTWARE ANALYSIS, DEBUGGING PROCESS, ERROR MESSAGING

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

IDE – Integrated development environment – Інтегроване середовище розробки;

OpenJDK – OpenJDK — вільна реалізація платформи Java;

GNU – вільна UNIX-подібна операційна система;

IntelliJ IDEA — комерційне інтегроване середовище розробки для різних мов програмування від компанії JetBrains.

ВСТУП

Актуальність дослідження. Інструменти аналізу програми застосовують алгоритми, такі як статичний аналіз, перевірка моделі та визначення типу, до вихідного коду, щоб допомогти розробникам виправити помилки компілятора, застосувати оптимізацію, виявити вразливі місця безпеки та обґрунтувати логіку програми. В інтегрованих середовищах розробки інструменти аналізу програм надають розробникам зворотній зв’язок щодо їх внутрішньої діагностики за допомогою повідомлень про помилки, використовуючи різноманітні текстові та візуальні презентації, такі як списки помилок, спливаючі підказки та вихідний код, підкреслений червоними завитками. Розробка зручних для людини повідомлень про помилки є важливою, оскільки повідомлення про помилки є основним каналом зв’язку, через який інструменти надають зворотній зв’язок розробникам.

Незважаючи на заплановану корисність цих інструментів, повідомлення про помилки, які створюють ці інструменти, є загадковими, неприємними та загалом некорисними для розробників, коли вони намагаються зрозуміти та вирішити повідомлення. Існуючі підходи до дослідження мов програмування намагалися висвітлити внутрішній процес міркування інструментів програмного аналізу та представити ці деталі розробникам, щоб допомогти їм у процесі розуміння. Однак ми стверджуємо, що орієнтована на інструмент перспектива простого розкриття деталей у повідомленні про помилку щодо внутрішніх алгоритмів інструментів недостатня: фундаментальна проблема полягає в тому, що обчислювальні інструменти не міркують про причини ідентифікованої помилки так само, як розробник, який намагається зрозуміти та реконструювати, чому інструмент спричинив цю конкретну помилку.

Процес налагодження зазвичай починається з визначення кроків для відтворення проблеми. Це може бути нетривіальним завданням, особливо з

паралельними процесами та деякими помилками. Конкретне середовище користувача та історія використання також можуть ускладнити відтворення проблеми.

Після відтворення помилки може знадобитися спростити введення програми, щоб полегшити її налагодження. Наприклад, помилка в компіляторі може спричинити його збій під час аналізу великого вихідного файлу. Однак після спрощення тестового випадку лише кілька рядків із вихідного вихідного файлу можуть бути достатніми для відтворення того самого збою. Спрощення можна зробити вручну за допомогою підходу «розділяй і володарюй», за якого програміст намагається видалити деякі частини вихідного тестового прикладу, а потім перевіряє, чи проблема все ще виникає. Під час налагодження в графічному інтерфейсі програміст може спробувати пропустити деякі дії користувача з початкового опису проблеми, щоб перевірити, чи достатньо решти дій для виникнення помилки.

Після того, як тестовий приклад достатньо спрощено, програміст може використовувати інструмент налагодження, щоб перевірити стани програми (значення змінних, а також стек викликів) і відстежити джерело проблеми (проблем). В якості альтернативи можна використовувати трасування. У простих випадках трасування — це лише кілька операторів друку, які виводять значення змінних у певні моменти під час виконання програми.

Інтерактивне налагодження використовує інструменти налагодження, які дозволяють виконувати крок за кроком виконання коду програми та призупиняти його для перевірки або зміни стану програми. Ці інструменти зазвичай підтримують точки спостереження, де виконання може тривати до зміни певної змінної, і точки перехоплення, які призводять до зупинки налагоджувача для певних видів програмних подій, таких як винятки або завантаження спільної бібліотеки.

Налагодження друку або трасування — це перегляд операторів трасування або операторів друку, які вказують на потік виконання процесу та прогресування даних. Трасування можна виконати за допомогою

спеціалізованих інструментів (наприклад, трасування GDB) або шляхом вставки операторів трасування у вихідний код. Останнє іноді називають налагодженням printf через використання функції printf у C. Цей вид налагодження вмикався командою TRON в оригінальних версіях мови програмування BASIC, орієнтованої на новачків. TRON означає "Trace On". TRON викликав друк номерів рядків кожного командного рядка BASIC під час виконання програми.

Віддалене налагодження — це процес налагодження програми, яка виконується в системі, відмінній від налагоджувача. Щоб розпочати віддалене налагодження, налагоджувач підключається до віддаленої системи через канал зв'язку, такий як локальна мережа. Потім налагоджувач може контролювати виконання програми у віддаленій системі та отримувати інформацію про її стан.

Пов'язані методи часто включають різні методи трасування, як-от перевірка файлів журналу, виведення стека викликів у разі збою та аналіз дампа пам'яті (або дампа ядра) процесу, що завершився збоєм. Дамп процесу може бути отриманий автоматично системою (наприклад, коли процес завершився через необроблену виняткову ситуацію), або за допомогою інструкції, вставленої програмістом, або вручну інтерактивним користувачем.

Мета і завдання дослідження. Метою кваліфікаційної роботи є розробка оптимальної інформаційної стратегії інтерпретації процесу відлагодження помилок для підвищення ефективності процесів дебаггінгу.

Для досягнення поставленої мети необхідно розв'язати такі задачі:

- виконати аналіз інструментів інтерпретації помилок під час процесу дебаггінгу;
- структурувати моделі представлення програмних помилок;
- розробити та імплементувати методи оптимізації та методології інтерпретації помилок в процесі дебаггінгу.

Об'єктом дослідження є процес дебагінгу та відлагодження програмного коду.

Предметом дослідження є методи, моделі та структура представлення програмних помилок з метою оптимізації процесів дебагінгу.

Методи дослідження базуються на використані методів формального аналізу процесу дебагінгу, методів об'єктно-орієнтованого програмування, методів побудови алгоритмів.

Наукова новизна одержаних результатів полягає у розробці концепції розширеного виведення пояснень про помилки, що дозволяє підвищити ефективність моделей та методів процесів дебагінгу.

Практичне значення одержаних результатів полягає в розробці оптимальної методології візуалізації повідомлень про помилки компілятора та методології інтерпретації помилок.

Апробація результатів дослідження. Матеріали дослідження було представлено у матеріалах І Всеукраїнської науково-практичної інтернет конференції “ІТ екосистема: цифровізація бізнес-процесів в умовах війни”, у тезах доповіді “Засоби підвищення ефективності локалізації помилок”.

Структура. Кількість розділів – 4. Загальний обсяг основної частини - 87 сторінок. Список використаних джерел містить – 52 позиції.

РОЗДІЛ 1. АНАЛІЗ ІНСТРУМЕНТІВ ІНТЕРПРЕТАЦІЇ ПОМИЛОК ПІД ЧАС ПРОЦЕСУ ДЕБАГГІНГУ

1.1 Дослідження програмних інструментів аналізу та інтерпретації помилок

Інструменти аналізу програми призначені для того, щоб допомогти розробникам виявити проблеми у вихідному коді: вони точно визначають небезпечну або небажану поведінку під час виконання, забезпечують відповідність специфікаціям мови програмування та позначають стилістичні проблеми, які заважають читабельності коду. Багато проблем, які виявляють інструменти програмного аналізу, також легко помітити під час перевірки вручну. На жаль, дослідження виявили, що вихідні програми інструментів аналізу — повідомлення про помилки — заплутані, неконструктивні, вводять в оману або незрозумілі. У результаті розробники витрачають непотрібні зусилля на розуміння та усунення дефектів, виявлених інструментами, або просто відмовляються від інших корисних інструментів, оскільки вони не можуть зрозуміти повідомлення про помилки.

Що робить ці повідомлення про помилки такими незрозумілими для розробників?

Інструменти аналізу програми повідомляють про ці проблеми розробникам через повідомлення про помилки, використовуючи різні текстові та візуальні презентації. Але замість того, щоб намагатися точно охарактеризувати, що таке повідомлення про помилку, давайте розглянемо п’ять конкретних прикладів, щоб дослідити різні аспекти того, чому розробникам може бути важко зрозуміти повідомлення про помилку. Багато систем досліджують текстові повідомлення про помилки, які можна знайти в середовищі консолі або терміналі, деякі адаптують текстові повідомлення про помилки до візуальних інтегрованих середовищ розробки (IDE) — таких як

Visual Studio Code. Зокрема, у цьому прикладі показано, як відповідність повідомлення про помилку залежить не лише від вмісту повідомлення, але й від того, як повідомлення про помилку розміщено в середовищі програмування.

Ми почнемо з компілятора C, для якого розробник намагається написати «Hello, world!». Ось вихідний код цієї програми:

```
1  #include <stdio.h>
2
3  int main() {
4    printf("Hello, world!\n")
5  }
```

Незважаючи на свою простоту, ця програма корисна для перевірки написання тексту "Привіт Світ!" який з’явитися на екрані і означає, що базові бібліотеки знаходяться в потрібному місці, і що вихідний код можна скомпілювати, виконати та успішно надіслати вихідні дані на консоль.

Однак цей вихідний список містить помилку: правила специфікації програмування C вимагають, щоб усі оператори закінчувалися крапкою з комою (;), а в коді відсутня одна крапка в кінці рядка 4. Аналіз програми в компіляторі визначає цю помилку і видає повідомлення про помилку:

```
hello.c, line 5: syntax error
hello.c, line 5: cannot recover from earlier errors: goodbye!
error: /usr/libexec/ccom terminated with status 1
```

Виявляється, це повідомлення про помилку не є технічно неправильним. Наприклад, стверджується, що помилка знаходиться в рядку 5, але здається, що помилка насправді повинна бути повідомлена в рядку 4. Незважаючи на це, помилка не надає жодних причин для того, щоб визначити її причину. Потім компілятор видає повідомлення goodbay! і статус завершення.

Чи справедливо використовувати компілятор, написаний у 1970-х роках, для ілюстрації незрозумілих повідомлень про помилки? Напевно ні.

Але це хороша основа для повідомлень про помилки. Незважаючи на свій інтерес, повідомлення про помилку містить мінімальні компоненти сучасних повідомлень про помилки: розташування, що вказує на проблему, hello.c, рядок 5 та опис проблеми. І багато рішень у C продовжують впливати на сучасні інструменти аналізу програм.

Давайте тепер розглянемо GCC, сучасний компілятор C, який є частиною GNU. Ось повідомлення про помилку для того самого «Hello, world!» в GCC:

```
hello.c: In function ‘main’:
hello.c:5:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

Компілятор GCC є вдосконаленням у порівнянні з попереднім: замість розпливчастої синтаксичної помилки він говорить нам, що крапка з комою є фактичним маркером, якого бракує. Інструмент аналізу програми також додає трохи кольору до повідомлення про помилку та деякий контекст щодо вихідного коду, наприклад, вказуючи, що проблема полягає у функції main.

Тим не менш, причина повідомлення про помилку дезорієнтує з точки зору розробників, оскільки інструмент видає місце, яке слідує за проблемою, а не місце, яке безпосередньо передує їй. Найпростіший спосіб проілюструвати це порівняти GCC з компілятором, який робить це правильно, принаймні в цьому випадку. Ось результати Clang, частини LLVM:

```
hello.c:4:28: error: expected ';' after expression
  printf("Hello, world!\n")
                           ^
                           ;
1 error generated.
```

Проблема тепер очевидна, і повідомлення про помилку також відповідає тому, як розробники думали про цю проблему. Іншими словами, «Мені не вистачає крапки з комою в кінці рядка 4» є більш прямим поясненням, ніж «Мені не вистачає крапки з комою безпосередньо перед

фігурною дужкою в рядку 5, але було б дивно додати кому двокрапка на початку рядка, тому вона має бути навіть перед цим. Компілятор насправді має позначати кінець рядка 4».

Синтаксичні проблеми, подібні до цих, неприємні для досвідчених розробників — навіть з оригінальною помилкою компілятора C – але це більше роботи, ніж нам доведеться робити. Однак новачків подібні повідомлення про помилки дуже бентежать. Міркування компілятора, суворо структуровані і можуть призвести до ідіосинкратичних, але, тим не менш, сумісних виправлень, як ось це:

- Компілятор Oracle Java (OpenJDK). Ось код на Java:

```
1  class Toy {
2      Toy() throws Exception { }
3  }
4
5  class Kite extends Toy {
6  }
```

- Дана частина коду призводить до такого повідомлення про помилку від компілятора OpenJDK:

```
Kite.java:5: error: unreported exception Exception in default constructor
class Kite extends Toy {
^
1 error
```

Давайте побудуємо обґрунтування того, що може викликати це повідомлення про помилку. По-перше, зауважимо, що повідомлення про помилку містить конструктор за замовчуванням, але такий конструктор явно не відображається в класі Kite. Можливо, саме тому маркер помилки ^ вказує на сам клас: конструктори за замовчуванням є неявними, тому немає конструктора, на який ми можемо вказати безпосередньо. Змінимо вихідний код, щоб явно додати цей конструктор і сподіватися, що це компілятор програми надати детальніші пояснення:

```
class Toy {
    Toy() throws Exception { }
}

class Kite extends Toy {
  Kite() { }
}
```

Знову компілюємо. Нове повідомлення про помилку не тільки вказує на інше місце, але й тепер розкриває більше інформації про проблему:

```
Kite2.java:6: error: unreported exception Exception;
must be caught or declared to be thrown
  Kite() { }
         ^
1 error
```

Таким чином, ми додаємо виняток throws у відповідь на це пояснення:

```
class Toy {
    Toy() throws Exception { }
}

class Kite extends Toy {
    Kite() throws Exception { }
}
```

Наша остання спроба прийнята компілятором, і тепер ми повинні знайти причини, чому. Наприклад, чому ми повинні були явно створити конструктор за замовчуванням ? Хіба вони не створюються автоматично, якщо ми не пишемо їх самі ? Виявляється, згідно зі специфікацією мови Java.

Таким чином, у випадку, коли конструктор за замовчуванням у суперкласі має пункт throws, компілятору заборонено автоматично генерувати конструктор за замовчуванням.

Тепер дослідимо на вихідний листинг у C++ під назвою ptrcopy.cpp , у якому ми будемо використовувати компілятор Clang від LLVM:

```
#include <iostream>
#include <memory>
#include <vector>

int main() {
    std::vector<std::unique_ptr<int>> foo;
    std::vector<std::unique_ptr<int>> bar = foo;
}
```

Метою цієї програми є певне копіювання вмісту foo до вмісту bar . Але спроба зробити це генерує помилку компілятора:

```
In file included from ptrcopy.cpp:2:
In file included from /usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../include/c++/7.2.0/memory:64:
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/stl_construct.h:75:38: error: call to deleted constructor of
      'std::unique_ptr<int, std::default_delete<int> >'
    { ::new(static_cast<void*>(__p)) _T1(std::forward<_Args>(__args)...); }
                                     ^   ~~~~~~~~~~~~~~~~~~~~~~~~~~~
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/stl_uninitialized.h:83:8: note: in instantiation of function template specialization
      'std::_Construct<std::unique_ptr<int, std::default_delete<int> >, const std::unique_ptr<int, std::default_delete<int> > &>'
      requested here
            std::_Construct(std::__addressof(*__cur), *__first);
                 ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/stl_uninitialized.h:134:2: note: in instantiation of function template specialization
      'std::__uninitialized_copy<false>::__uninit_copy<__gnu_cxx::__normal_iterator<const std::unique_ptr<int,
      std::default_delete<int> > *,
      std::default_delete<int> >, std::allocator<std::unique_ptr<int, std::default_delete<int> > > >, std::unique_ptr<int, std::default_delete<int> > *>' requested here
    { return std::uninitialized_copy(__first, __last, __result); }
                  ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/stl_vector.h:331:9: note: in instantiation of function template specialization
      'std::__uninitialized_copy_a<__gnu_cxx::__normal_iterator<const std::unique_ptr<int, std::default_delete<int> > *,
      std::vector<std::unique_ptr<int,
      std::default_delete<int> >, std::allocator<std::unique_ptr<int, std::default_delete<int> > > >, std::unique_ptr<int, std::default_delete<int> > *,
      std::unique_ptr<int, std::default_delete<int> > >' requested here
          std::__uninitialized_copy_a(__x.begin(), __x.end(),
               ^
ptrcopy.cpp:7:43: note: in instantiation of member function 'std::vector<std::unique_ptr<int, std::default_delete<int> >, std::allocator<std::unique_ptr<int,
      std::default_delete<int> > > >::vector' requested here
  std::vector<std::unique_ptr<int>> bar = foo;
                                          ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/bits/unique_ptr.h:388:7: note: 'unique_ptr' has been explicitly marked deleted here
      unique_ptr(const unique_ptr&) = delete;
      ^
1 error generated.
```

Дивлячись на це повідомлення про помилку, можемо отримати підказки, які допоможуть реконструювати пояснення справжньої проблеми. Наприклад, можна просканувати повідомлення про помилку, щоб побачити, що файли, які цікавлять, це пам’ять (рядок 2), stl_construct.h (рядок 3), stl_uninitialized.h (рядок 7, рядок 11 і рядок 17), stl_vector.h (Рядок 22) і список джерел, який ми написали, ptrcopy.cpp (Рядок 28). Оскільки єдиним файлом є ptrcopy.cpp, можна зробити висновок, що розташування помилок представлено в зворотному порядку: щось відбувається в пам’яті, і ця проблема з’являється через різні файлів C++, доки ми не повернемося до ptrcopy . Хоча проблема не виникає, строго кажучи, доки ми не досягнемо пам’яті, не дуже корисно отримати проблему, визначену в бібліотеці, яку ми навіть не писали.

Що, якби аналіз програми замість цього показав повідомлення про помилку з точки зору розробника, а не з точки зору компілятора ? Ось приклад того, як таке повідомлення може виглядати:

```
ptrcopy.cpp: cannot construct 'bar' from 'foo':
foo's template type is non-copyable
    std::vector<std::unique_ptr<int>> bar = foo;
                                            ^
```

Дана версія повідомлення про помилку є цікавим контрастом з кількох причин. По-перше, він точно визначає місце, яке насправді написав розробник, по-друге, це аргументує презентацію, яка багато в чому менш точна, ніж оригінальне повідомлення про помилку, але набагато легша для читання. По-третє, навіть із цією втратою точності, очевидно, у чому полягає проблема: unique_ptr не можна скопіювати. Спроба призначити вектор unique_ptr іншому вектору означатиме, що десь у вихідному коді вектора обов’язково потрібно буде скопіювати унікальний покажчик.

Поки що ми розглядали лише текстові повідомлення про помилки. Однак ми стверджуватимемо, що відповідність повідомлення про помилку залежить не лише від вмісту повідомлення, а й від того, як повідомлення про помилку розміщено в середовищі програмування, наприклад, у терміналі чи в

IDE. Тобто іноді «носієм є повідомлення», і саме середовище формує спосіб, у який ми повинні надавати повідомлення про помилки розробнику.

Давайте розглянемо приклад цього за допомогою ESLint, підключаємої утиліти пошуку помилок для JavaScript. Під можливістю підключення ми маємо на увазі:

- розробники можуть додавати спеціальні правила пошуку помилок, щоб розширити можливості інструменту;
- інструмент призначений для об'єднання та використання в інших робочих процесах, таких як інтегровані середовища розробки та системи побудови. Таким чином, інструмент підтримує кілька вихідних форматів або носіїв для представлення.

Дослідимо, що відбувається, коли ми застосовуємо ESLint до наступного файлу JavaScript:

```
1  const who = 'Titus';
2  console.log('Hello, ' + who + '!');
```

Залежно від форматування за замовчуванням це призводить до того, що ESLint видає таке повідомлення про помилку:

```
hello.js: line 2, col 13, Error - Unexpected string concatenation.
↪  (prefer-template)
```

Ми можемо використовувати літерал шаблону щоб вирішити повідомлення про помилку:

```
1  const who = 'Titus';
2  console.log(`Hello, ${who}!`);
```

На відміну від повідомлень про помилки від OpenJDK, GCC і LLVM, повідомлення про помилку від ESLint виглядає майже як повернення до більш ранньої ери компіляторів. На відміну від будь-якого з цих інструментів, повідомлення про помилку ESLint не надає контекстного фрагмента коду. На

відміну від GCC і LLVM, ESLint не розфарбовує вивід, а prefer-template виглядає як внутрішній код помилки, який не допомагає розробнику.

Якби ми зосереджувалися лише на середовищі консолі, ми могли б розглянути відносно прості вдосконалення цього повідомлення про помилку, щоб розширити запропоноване виправлення шаблону та додати обґрунтування помилки:

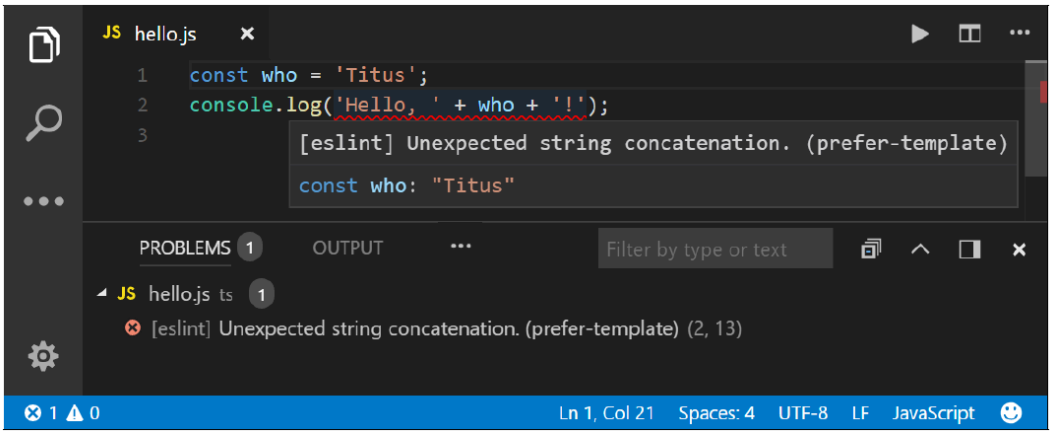
```
× [eslint] Unexpected string concatenation (2, 13)
           Fix:  Use template literals instead of concatenation.
                 (re-run with --fix to automatically fix this problem)
           Why?  Template literals give you a readable, concise syntax
                 with proper newlines and string interpolation features.
                 (see: airbnb.io/javascript/#es6-template-literals)
```

Наведене вище повідомлення про помилку замінює ключ шаблону природною мовою. Відповідно, це конкретне виправлення тривіально виправлено ESLint, тому повідомлення також пропонує доступність цього автоматичного відновлення. Враховуючи причину помилки, можна побачити, що причина, чому ESLint рекомендує цю зміну, полягає в тому, що стандарт JavaScript Airbnb вважає літерали шаблонів більш читабельними. Розширене пояснення пропонується за посиланням і надає приклади перетворення операцій конкатенації рядків у шаблонні літерали.

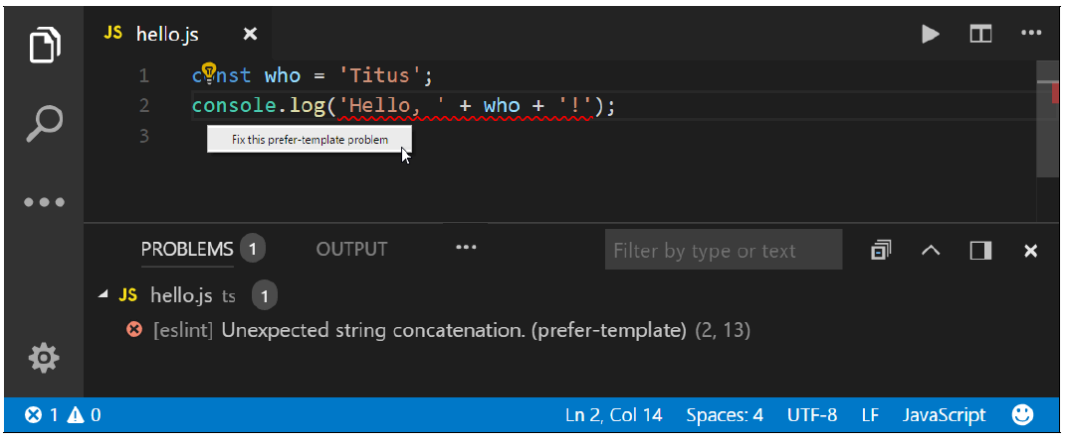
1.2 Область застосування інструментів інтерпретації помилок

Незважаючи на те, що ESLint підтримує консольний вихід, ESLint насправді призначений для використання як будівельного блоку для аналізу програм, який буде включено в інші середовища програмування та збірки. Щоб проілюструвати це, давайте знову розглянемо форматування за замовчуванням, але цього разу відтворене в коді Visual Studio (рис. 1.1). На рисунку 1.1а ми бачимо, що ESLint представляє своє повідомлення про помилку як спливаючу підказку, коли розробник наводить курсор на червоне хвилясте підкреслення. Крім того, розробник може двічі клацнути

повідомлення про помилку на панелі проблем, щоб перейти безпосередньо до рядка 2 hello.js . Враховуючи однорядкові можливості на панелі проблем, тепер має сенс надати стисле повідомлення про помилку, яке розробник може використовувати для швидкого переходу до відповідного контексту коду. Подібним чином, (prefer-template) текст у повідомленні про помилку за замовчуванням тепер стає інтерактивним доступом, який дозволяє розробнику автоматично відновлювати код у своїй IDE (Рис. 1.1b). Тому надання контекстного фрагмента коду в самому повідомленні про помилку буде зайвим, оскільки помилки будуть представлені в IDE.



а) Ідентифікація помилки



б) Виправлення помилки

Рисунок 1.1 – Представлення помилки «неочікуваної конкатенації рядків» в IDE Visual Studio Code

На даному рисунку 1.1 піктограма, накладання спливаючої підказки та червоне хвилясте підкреслення, надають додатковий контекст для розробника.

Враховуючи повідомлення про помилки, які ми детально описали, можна стверджувати, що виникають труднощі, з якими стикаються розробники при появі таких повідомленнями.

Метою даної роботи є оцінка системи, яка застосовує теорію раціональної реконструкції до повідомлень про помилки як єдине пояснення того, чому розробникам важко зрозуміти повідомлення про помилки. Значення даного дослідження надають додатковий контекст для розробника.

Повідомлення про помилки як раціональні конструкції є більш корисними для розробників, ніж базові повідомлення про помилки, оскільки раціональні конструкції узгоджуються способом, у який вони виробляють стратегію та міркують про проблеми у своєму коді за наявності помилок. Крім того, розробка зручних для людини повідомлень про помилки є важливою, оскільки повідомлення про помилки є основним каналом зв’язку, через який інструменти надають зворотній зв’язок розробникам. Отже, дане дослідження є актуальним і використовує систематичну, теоретичну призму для дослідження повідомлень про помилки в інструментах програмного аналізу та втілює шість постулатів:

Постулат I — повідомлення про помилки є звичайними. Вибрані приклади — це повсякденні повідомлення, які заплутують розробників, взяті з інструментів аналізу програм, які використовуються в промисловості. Як правило, типи повідомлень про помилки, з якими стикаються розробники, не вимагають від них знання концепцій мови, а також повідомлення про помилки не виникають через надзвичайні обставини.

Постулат II. Повідомлення про помилки не є помилковими. Помилкові спрацьовування є відомою проблемою інструментів програмного аналізу, особливо інструментів статичного аналізу, які використовують апроксимації, щоб визначити, чи існує проблема. Але жоден із вибраних прикладів не був

помилковим, і всі вони вказували на реальну проблему в коді. Це свідчить про те, що розробники мають труднощі з повідомленнями про помилки, навіть якщо вони виявляють реальну проблему в коді.

Постулат III. Розробники є експертами середнього рівня. Це проблема не лише для новачків. Тобто не новачки в програмуванні також стикаються з такими проблемами.

Постулат IV. Існує не тільки один спосіб автоматичного відновлення програми. Навіть невеликі програми мають комбінаторно великий простір проектування для програмних перетворень, які б видалили б повідомлення про помилку компілятора. Щоб проілюструвати, альтернативний спосіб зробити «Hello, world!» синтаксично дійсним для прикладів у попередньому пункті необхідно просто видалити оператор printf. Таке виправлення здається неприпустимим, якщо розробник має намір надрукувати рядок на консолі, але цілком логічним, якщо розробник мав на меті «Hello, world!» і це лише служить відправною точкою для програми, яку вони фактично мали намір реалізувати. Хороші автоматичні виправлення корисні як прискорювачі для розробників, але вони не позбавляють необхідності розуміти повідомлення про помилку.

Постулат V. Повідомлення про помилки представляють недостатню структуру проблеми. Повідомлення про помилки представляють лише ознаки проблеми, а розробник повинен знайти обґрунтування, чому виникає проблема. Було наведено вище, у прикладі неповідомленої виняткової ситуації для Java, де розробнику потрібно було реконструювати ланцюжок причин, щоб визначити причину помилки.

Постулат VI. Повідомлення про помилки раціональні, але лише з точки зору компілятора. Давайте знову розглянемо вихідний код для “Hello, world!” приклад, але цього разу переформатуємо цого наступним чином:

```
int main(){printf("Hello, world!")}
```

Деякі компілятори видаляють незначні пробіли, оскільки вони непотрібні в конвеєрі аналізу програми і це призводить до програми, яка виглядає інструментом, по суті, як наведена вище. З точки зору компілятора, очікуване повідомлення про помилку ';' перед '}' маркер тепер цілком раціональний, якщо ми переорієнтуємося на перспективу того, як компілятор думає про ситуацію. Але розробникам не потрібно знати внутрішню логіку інструментів аналізу програм, щоб зрозуміти їхні повідомлення про помилки.

Перші три постулати усувають конкретні фактори як причину труднощів розробника з повідомленнями про помилки: розробники відчувають труднощі у випадках, коли повідомлення про помилки є звичайними, коли вони не є хибними спрацьовуваннями і навіть коли розробники добре володіють мовами програмування та інструментами.

Четвертий постулат передбачає, що повідомлення про помилки корисні розробникам для перевірки, навіть якщо інструменти пропонують автоматичні виправлення. Але саме п’ятий і шостий постулати спонукають теорію досліджувати: ці два постулати припускають невідповідність між тим, як інструменти аналізу програм передають розробникам повідомлення про помилки, і тим, як розробники намагаються зрозуміти проблему.

1.3 Теоретичні основи дослідження

Розглянемо сценарій, у якому розробник Josh стикається з незрозумілим повідомленням про помилку. Він не може зрозуміти повідомлення про помилку, тому звертається за допомогою до колеги Еммі. Подібно до того, як ми робили для прикладів у попередньому пункті, Еммі пояснює повідомлення про помилку, пропонуючи причини та обґрунтування, щоб продемонструвати, що проблема насправді є проблемою. Можна зобразити цей сценарій так, як показано на рисунку 1.2, де Еммі буде співрозмовником — учасником дискурсу.

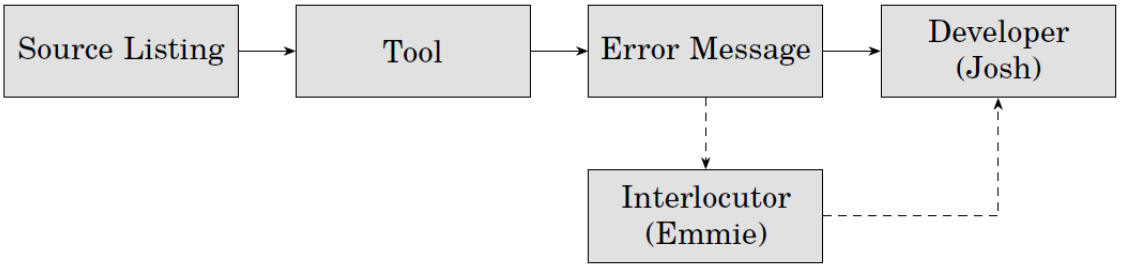


Рисунок 1.2 – Теоретичний фреймворк раціональної реконструкції

Що сказала Еммі, на повідомлення про помилку в інструменті аналізу програми, що дозволило Джошу зрозуміти проблему? Загалом, чим взаємодія між людьми відрізняється від взаємодії між людиною та комп'ютером, і чи можуть ці відмінності пояснити, чому повідомлення про помилки в інструментах аналізу програм створюють проблеми для розробників?

Теорія яка застосовується для дослідження труднощів розробника з повідомленнями про помилки — це раціональна реконструкція. Обґрунтування — це набір причин чогось, в даному випадку це повідомлення про помилку, створене інструментом аналізу програми. Якщо повідомлення про помилку є неповноцінним і не містить достатнього обґрунтування, Джош повинен сам реконструювати обґрунтування повідомлення про помилку та дійти того самого висновку, що й у повідомленні про помилку.

Процес виявлення цих обґрунтувань є раціональною реконструкцією. У рамках цієї теорії обґрунтування мають дві форми:

- як обґрунтування-пояснення (називається аргументом), у якому обґрунтування функціонує як доказ, що підтримує висновок;
- наслідок-пояснення, в якому обґрунтування функціонує як причина для висновку.

Раціональна реконструкція є корисною теоретичною основою для застосування до повідомлень про помилки з чотирьох причин:

Раціональна реконструкція — це процес, зосереджений навколо звичайного людського діалогу. Виходячи з лінгвістики, раціональна реконструкція застосовує орієнтовану на людину призму як засіб для

побудови інтуїтивних, але логічних пояснень, призначених для споживання людьми, як обмін між Еммі та Джошем. Але чи є взаємодія між людьми основною істиною для взаємодії людини з комп'ютером? Впливова теорія комп'ютерів як соціальних акторів припускає, що це так: люди ставляться до комп'ютерів і реагують на комп'ютери так, ніби вони справжні люди. А подальші дослідження стверджують, що для обчислювальних агентів часто важливо імітувати те, як люди поводяться під час взаємодії між людьми, і наближати їх у взаємодії між людьми. Раціональні реконструкції допускають ортогональні пояснення проблеми, і наслідком дослідження є те, що можна розглядати представлення повідомлення про помилку як незалежне від внутрішнього функціонування інструменту. Еммі не потрібно знати внутрішню частину інструменту аналізу програм, щоб надати Джошу достатнє пояснення, чому видається повідомлення про помилку.

Раціональна реконструкція з деяким успіхом застосовувалася в інших областях розробки програмного забезпечення. Наприклад, процес проектування Парнаса та Клементса пропонує, щоб документація з розробки програмного забезпечення виглядала так, ніби вона була розроблена за допомогою точного процесу вимог, навіть якщо насправді продукт не проектується таким чином. Іншими словами, конструкторська документація програмного забезпечення є раціональною реконструкцією.

Висновки до розділу 1

В даному розділі проведено основні труднощі з інтерпретацією повідомлень про помилки, створені інструментами програмного аналізу, що відповідно є значною причиною нездатності розробників усунути дефекти: труднощі з інтерпретацією повідомлень про помилки можна пояснити тим, що повідомлення про помилки визначають як недостатню раціональну реконструкцію як у візуальних, так і в текстових представленнях. Переважно, розробники віддають перевагу повідомленням про помилки з

правильною структурою аргументів, а не недоліком аргументів, але віддадуть перевагу недолікам аргументів, якщо вони забезпечують вирішення проблеми. Також в розділі описано контекст проблеми дослідження, сформульована мета та значення роботи, а також пропонується теоретична основа раціональної реконструкції, за допомогою якої відбувається дослідження повідомлення про помилки.

РОЗДІЛ 2. ОПТИМАЛЬНІ МОДЕЛІ ТА СТРУКТУРА ПРЕДСТАВЛЕННЯ ПРОГРАМНИХ ПОМИЛОК

2.1 Дослідження інструментів аналізу програмних помилок

Інструменти аналізу програм відносяться до широкого класу програмного забезпечення, призначеного для допомоги розробникам, з додатками через архітектуру програмного забезпечення, розуміння програми, еволюцію програми, тестування, версії програмного забезпечення та перевірку.

Аналіз вихідного коду — це процес вилучення інформації про програму з її вихідного коду або артефактів (наприклад, з байт-коду Java або результатів виконання), згенерованих із вихідного коду за допомогою автоматичних інструментів.

Вихідний код — це будь-який статичний, текстовий, зрозумілий людині, повністю виконуваний опис комп’ютерної програми, який може бути автоматично скомпільований у виконувану форму. Для підтримки динамічного аналізу опис може містити документи, необхідні для виконання або компіляції програми, наприклад, вхідні дані програми.

Перший вимір – коли виконується аналіз програми. У статичному аналізі інструменти перевіряють вихідний код — або безпосередньо, або як абстракцію вихідного коду — і виявляють дефекти в коді без виконання програми. У динамічному аналізі інструменти інструментально або іншим чином перевіряють час виконання та аналізують потік виконання, щоб виявити дефекти.

Статичний і динамічний аналіз також можна застосувати синергетично для посилення програмного аналізу. Як приклад гібридного аналізу, Check’n’Crash поєднує статичне підтвердження теореми з виконанням

динамічного тесту, щоб усунути хибні попередження та покращити легкість розуміння повідомлень про помилки через створення прикладів Java.

У другому вимірі виконується аналіз програми. Наприклад, FindBugs є автономним інструментом статичного аналізу для Java, який доповнює повідомлення про помилки, надані компілятором Java. Навпаки, LLVM вбудовує інструменти аналізу програми безпосередньо в конвеєр компіляції.

На жаль, жоден із цих параметрів реалізації не є задовільним для розуміння труднощів розробника в рамках теоретичної основи розробки. У раціональній реконструкції деталь реалізації того, чи інформація для проблеми отримана за допомогою статичного, динамічного або гібридного аналізу, ортогональна відповідному представленню повідомлення про помилку.

Наприклад, хоча і LLVM (статичний) може повідомляти про неініціалізовані змінні, раціональна реконструкція припускає, що тип аналізу не повинен диктувати відповідне повідомлення про помилку для представлення розробнику.

2.2 Текстові представлення помилок під час виконання програмного аналізу

Розглянемо текстові представлення програмного аналізу. Знайома схема повідомлень про помилки в інструментах аналізу програм складається з розташування, яке вказує, де виникає проблема, що вказує на те, що пішло щось не так, і деякої додаткової інформації, як-от серйозність проблеми, код помилки, підказки щодо вирішення, і контекст коду, щоб допомогти розробнику вирішити проблему.

Спочатку розпочнемо дослідження цієї схеми повідомлення про помилку, використовуючи наступні три переліки вихідного коду, щоб викликати помилку під час аналізу програми:

1 The Java implementation:

```
class Brick {
    void m(int i, double d) { }
    void m(double d, int m) { }

    {
      m(1, 2);
    }
}
```

2 The C# implementation:

```
namespace Program {
  class Brick {
      void m(int i, double d) { }
      void m(double d, int m) { }

    static int Main(string[] args) {
      var b = new Brick();
      b.m(1, 2);
      return 0;
    }
  }
}
```

3 The C++ implementation:

```
class Brick {
    void m(int i, double d) { }
    void m(double d, int m) { }
};

int main() {
```

У всіх трьох лістингах — Java, C# та C++ було введено неоднозначну помилку методу: незважаючи на різні сигнатури типів, аналіз програми не може усунути неоднозначність, яку з двох реалізацій-кандидатів, якщо викликати метод m(1,2).

Одним із можливих виправлень є явно вказати тип аргументу як m((int)1,(double)2) (у Java) або bm((int)1, (double)2) (у C# та C+).

Потім вводимо ці списки в різні компілятори, щоб отримати повідомлення про помилки. Для повідомлень про помилки Java ми використовуємо OpenJDK і компілятор Eclipse. Для C# використовуємо Roslyn і Mono, для C++ відповідно використовуємо LLVM/Clang і GCC.

Вибір компіляторів ілюструє різноманіття того, як інструменти обирають представлення повідомлень про помилки для концептуально однакової проблеми.

Застосування вихідного коду до відповідних інструментів призводить до таких повідомлень про помилку під час компіляції:

1 OpenJDK (Java):

```
Brick.java:6: error: reference to m is ambiguous
      m(1, 2);
      ^
  both method m(int,double) in Brick and method m(double,int) in Brick
    ↪ match
1 error
```

2 Eclipse (Java)

```
----------
Brick.java (at line 6)
       m(1, 2);
       ^
The method m(int, double) is ambiguous for the type Brick
----------
1 problem (1 error)
```

3 Roslyn (C#)

```
Brick.cs(8,9): error CS0121: The call is ambiguous between the
  ↪ following methods or properties: 'Brick.m(int, double)' and
  ↪ 'Brick.m(double, int)'
```

4 Mono (C#)

```
Brick.cs(8,9): error CS0121: The call is ambiguous between the
  ↪ following methods or properties: `Program.Brick.m(int, double)' and
  ↪ `Program.Brick.m(double, int)'
Brick.cs(3,12): (Location of the symbol related to previous error)
Brick.cs(4,12): (Location of the symbol related to previous error)
Compilation failed: 1 error(s), 0 warnings
```

5 GCC (C++)

```
Brick.cpp: In function ‘int main()’:
Brick.cpp:8:13: error: call of overloaded ‘m(int, int)’ is ambiguous
     b.m(1, 2);
            ^
Brick.cpp:2:10: note: candidate: void Brick::m(int, double)
     void m(int i, double d) { }
          ^
Brick.cpp:3:10: note: candidate: void Brick::m(double, int)
     void m(double d, int m) { }
          ^
```

6 LLVM (C++)

```
Brick.cpp:8:7: error: call to member function 'm' is ambiguous
    b.m(1, 2);
    ~~^
Brick.cpp:2:10: note: candidate function
    void m(int i, double d) { }
         ^
Brick.cpp:3:10: note: candidate function
    void m(double d, int m) { }
         ^
1 error generated.
```

Є кілька відмінностей у звітності цих інструментів в описі проблеми, допоміжному контексті та форматуванні. OpenJDK, Eclipse і Roslyn вказують місце виклику, але не розташування методів-кандидатів. Крім того, повідомлення про помилку Eclipse також не вказує, для яких методів виклик є неоднозначним. Це на відміну від GCC, LLVM і Mono, що повідомляє про виклик, кандидатів і місцеположення. Однак GCC і LLVM повідомляють про іншу позицію стовпця для помилки: GCC вказує на проблему в кінці методу call(8:13), тоді як LLVM вказує на проблему bname, b.m(8:7). Існують інші невеликі варіації в тому, як рядок і стовпець представлені користувачу.

OpenJDK, Eclipse, GCC і LLVM вводять фрагменти безпосередньо з вихідного коду у звіт про помилку, щоб надати розробнику контекст; Roslyn і Mono ні. Відповідно Mono, вказує, що він не знає, чи є m методом чи властивістю. Нарешті, GCC і LLVM розфарбовують вихідні повідомлення про помилки для кращої читабельності. Інструментарії LLVM стверджують, що кольори полегшують розрізнення різних елементів повідомлення про помилку.

Як інструменти аналізу програми створюють ці повідомлення про помилки? Повідомлення, які було описано, виводяться з шаблону діагностики. Шаблони — це рядкові літерали, які дозволяють вставляти в них вирази або інтерполювати їх. Наприклад, у OpenJDK рядок опису помилки міститься у файлі compiler.properties:

```
compiler.properties:
   # 0: name, 1: symbol kind, 2: symbol, 3: symbol, 4: symbol kind, 5:
    ↪  symbol, 6: symbol
   compiler.err.ref.ambiguous=\
       reference to {0} is ambiguous\n\
       both {1} {2} in {3} and {4} {5} in {6} match
```

Параметри, які можна інтерполювати, позначаються як 0, 1, 2 і так далі. Реалізація Roslyn використовує подібну схему інтерполяції через файл XML під назвою CSharpResources.resx :

```
<data name="ERR_AmbigCall" xml:space="preserve">
    <value>The call is ambiguous between the following methods or
     ↪  properties: '{0}' and '{1}'</value>
</data>
```

Реалізація Roslyn підтримує лише два параметри, як видно в представленому повідомленні про помилку.

Рядок шаблону та пов’язані метадані об’єднуються як діагностичні об’єкти та передаються форматеру в інструменті аналізу програми. Форматувальник доповнює рядок шаблону інформацією метаданих, наприклад розташування. Форматувальник також розфарбовує вихідні дані та включає відповідні фрагменти вихідного коду, якщо ці можливості доступні у реалізації форматера.

Опишемо форму шаблонних повідомлень, яка регулярно є складною для розуміння — це приховані «поля з синтаксичними помилками». Синтаксична помилка виникає, коли аналіз програми стикається з неочікуваним маркером, наприклад, із неправильно розставлених крапок з комою, зайвих або відсутніх фігурних дужок або оператора case без охоплюючого перемикача. Зокрема, одним із недоліків повідомлень про синтаксичні помилки є те, що інструменти аналізу програм іноді повідомляють про розташування синтаксичної проблеми, далеко від фактичної причини синтаксичної помилки. Наприклад, розглянемо таку програму Racket, яка реалізує факторіал:

```
1  #lang racket
2
3  (define (factorial n)
4    (if (= n 0) 1
5        (* n (factorial (- n 1)))))
```

Ця програма реалізує правильну поведінку, але має синтаксичну помилку через відсутність закриваючої дужки. Racket повідомляє про цю синтаксичну помилку в рядку 3:

```
factorial.rkt:3:0: read: expected a `)' to close `('
```

Помилка насправді правильна, оскільки немає відповідної закриваючої дужки для відкриваючої дужки, що починається в рядку 3. Однак, щоб фактично усунути дефект, потрібно додати закриваючу дужку в кінці рядка 5:

```
5        (* n (factorial (- n 1))))
```

Існує багато досліджень щодо усунення синтаксичних помилок — широко висвітлених у різноманітних оглядах літератури [42, 44] — які відбувалися в тандемі з розробкою навіть найперших компіляторів. Відмі дослідження, де описують деякі з ранніх схем відновлення та виправлення синтаксичних помилок і характеризують компроміси в реалізації різних схем. Наприклад, одному з найперших і найпростіших методів усунення помилок, аналізатор видаляє вхідні маркери, доки аналізатор не виявить маркер, який дозволить йому продовжити обробку вихідного коду. Хоча цей підхід легко реалізувати, повідомляється, що він призводить до некорисних повідомлень про помилки через відсутність інформації, доступної на момент повідомлення про помилку, а такий підхід також призводить до помилкових повідомлень про помилки. Таким чином, наступні підходи були зосереджені на:

- виявленні помилок або зменшенні різниці між місцем виявлення помилки та точкою, де вона фактично виникає;

- виправленні помилок, тобто наданні розробнику одного або кількох варіантів виправлення які перетворюють неправильне введення в синтаксично правильне.

На відміну від методів, які математично визначають поняття мінімізації відстані до помилки, існує модель природної мови покращення звітування про помилки, стверджуючи, що люди читають код, так само як природну мову. Також мовні моделі можуть успішно знаходять та виправляють синтаксичні помилки в коді, написаному людиною, без формального аналізу.

Ще інші підходи досліджували зменшення впливу щодо створення більш корисних повідомлень про помилки. Наприклад, існує інструмент під назвою Merr — генератор метапомилок, який дозволяє розробникам компіляторів пов’язувати створені вручну діагностичні повідомлення з синтаксичними помилками. З цієї специфікації помилок і пов’язаного з ними повідомлення Merr визначає відповідні стани аналізу та вхідний маркер і вставляє функцію помилки в синтаксичний аналізатор, щоб створити повідомлення про помилку у відповідній точці. Також є вдосконалений підхід, який дозволяє синтаксичному аналізатору автоматично створювати колекцію помилкових тверджень замість того, щоб користувач надавав приклади вручну. Незважаючи на те, що обидва підходи корисні, вони все одно вимагають від вручну написати супровідне повідомлення про помилку, щоб охопити цю діагностичну інформацію.

2.3 Концепція розширеного виведення пояснень про помилки

Інструменти аналізу програми можуть надавати додаткові канали для розширених пояснень про повідомлення про помилку, як альтернативу стислим, рядково-орієнтованим презентаціям повідомлень про помилки. Стислі повідомлення про помилки в інструментах програмного аналізу не надають достатньо інформації, однак розробники не хочуть переглядати великі описи помилок, щоб виявити проблему.

Додатковий механізм надання розширених пояснень може узгодити ці суперечливі вимоги. Розширені пояснення також можуть допомогти зрозуміти, коли розробник не знає про причину помилки концепції в повідомленні і дозволяють їм вибірково досліджувати незнайомі повідомлення про помилки з довшим, більш детальним поясненням.

Щоб продемонструвати, наскільки розширені пояснення корисні на практиці, розглянемо спрощений конвеєр збірки з використанням Bazel — випуску системи збірки з відкритим вихідним кодом, яка використовується внутрішньо в Google. Bazel містить інструмент аналізу програми під назвою Error Prone, який визначає дефекти коду Java.

Конвеєр збирання керується файлом BUILD , який визначає цілі для системи збирання:

```
java_library(
    name = "shortset",
    srcs = ["ShortSet.java"],
)
```

У цьому сценарії присутня лише одна ціль збірки під назвою shortset і ця ціль збірки має скомпілювати лише один файл Java, ShortSet.java. Ось файл Java:

```
1  import java.util.Set;
2  import java.util.HashSet;
3
4  public class ShortSet {
5    public static void main (String[] args) {
6        Set<Short> s = new HashSet<>();
7        for (short i = 0; i < 100; i++) {
8          s.add(i);
9          s.remove(i - 1);
10       }
11
12       System.out.println(s.size());
13   }
14 }
```

Створення shortest target за допомогою Bazel призводить до такого результату, який містить повідомлення про помилку:

```
INFO: Analysed target //:shortset (0 packages loaded).
INFO: Found 1 target...
ERROR: /BUILD:1:1: Building libshortset.jar (1 source file) failed (Exit
 ↪  1)
ShortSet.java:9: error: [CollectionIncompatibleType] Argument 'i - 1'
should not be passed to this method; its type int is not compatible with
its collection's type argument Short
        s.remove(i - 1);
                 ^
    (see http://errorprone.info/bugpattern/CollectionIncompatibleType)
Target //:shortset failed to build
Use --verbose_failures to see the command lines of failed build steps.
INFO: Elapsed time: 0.582s, Critical Path: 0.29s
FAILED: Build did NOT complete successfully
```

Повідомлення про помилку вбудовано в інші вихідні дані збірки та починається в рядку 3 і закінчується в рядку 9, відповідно повідомлення про помилку також відносно стисле. Однак повідомлення про помилку вказує на зовнішню документацію, яка докладно пояснює причину цього повідомлення, чому навряд чи видалення дійсно видалить елемент і чому лише система типів у Java не може виявити цю проблему.

Делегування цього пояснення зовнішньому джерелу зменшує багатослівність вихідних даних збірки, але все ще дозволяє розробнику отримати доступ до додаткових пояснень щодо повідомлення. Якщо розробник уже знайомий з повідомленням про помилку, то йому взагалі не знадобиться розширене пояснення.

З цієї точки зору Error Prone є модернізованою реалізацією ранніх експертних систем, таких як експертна система для налагодження програм COBOL. У реалізації COBOL розробник міг взяти код помилки, як -от 3A13 (відсутня крапка після пропозиції VALUE), і отримати код помилки з бази даних, щоб отримати детальне пояснення, яке включало поради щодо вирішення проблеми.

Обмеження підходу, який використовує Error Prone, полягає в тому, що розширене пояснення відокремлено від контексту коду розробників. Ось фрагмент розширеного пояснення:

In a generic collection type, query methods such as `Map.get(Object)` and `Collection.remove(Object)` accept a parameter that identifies a potential element to look for in that collection. This check reports cases where this element *cannot* be present because its type and the collection's generic element type are "incompatible." A typical example:

```
Set<Long> values = ...
if (values.contains(1)) { ... }
```

This code looks reasonable, but there's a problem: The `Set` contains `Long` instances, but the argument to `contains` is an `Integer`.

Іншими словами, розробник повинен оцінити приклад, відмінний від коду, який він насправді написав, щоб зрозуміти пояснення. Навіть якщо розробник розуміє приклад, наведений у розширеному поясненні, він все одно може не зрозуміти, як проблема стосується його власного коду.

Щоб пом'якшити цю проблему, компілятор Dotty пропонує прапорець -- explain , який може надати пояснення, яке є контекстним для коду, який фактично написав розробник. Наприклад, розглянемо наступний фрагмент коду Dotty (мова Dotty є надмножиною Scala):

```
1 try {
2   foo()
3 }
```

За замовчуванням цей фрагмент коду створює таке повідомлення про помилку від компілятора Dotty:

```
-- [E002] Syntax Warning: scala.test ----------------------------------
1 | try {
  | ^
  | A try without catch or finally is equivalent to putting
  | its body in a block; no exceptions are handled.
2 |   foo()
3 | }
```

Подібно до мов програмування Rust і Elm, зручність використання повідомлень про помилки та інструментів є однією з цілей розробки Dotty.

Отже, це повідомлення про помилку вже досить ефективно працює: воно описує розташування помилки в контексті коду, описує, що механізм спроби буде робити в цьому контексті, і надає обґрунтування, в чому це

проблема, оскільки є ймовірність, що розробник новачок та не розуміє, що насправді робить конструкція try. Потім вони можуть передати прапорець --explain Dotty, щоб отримати розширене пояснення:

```
1  Explanation
2  ===========
3  A try expression should be followed by some mechanism to handle any
4  exceptions thrown. Typically a catch expression follows the try and
5  pattern matches on any expected exceptions. For example:
6
7  import scala.util.control.NonFatal
8
9  try {
10   foo()
11 } catch {
12   case NonFatal(e) => ???
13 }
14
15 It is also possible to follow a try immediately by a finally - letting


16 the exception propagate - but still allowing for some clean up in
17 finally:
18
19 try {
20   foo()
21 } finally {
22   // perform your cleanup here!
23 }
24
25 It is recommended to use the NonFatal extractor to catch all exceptions
26 as it correctly handles transfer functions like return.
```

На відміну від розширеного пояснення від Error Prone, пояснення від Dotty містить код, написаний розробником. Це легко побачити з використання foo (рядок 10 і рядок 20).

Поточна реалізація цього в Dotty є досить рудиментарною та лише трохи складнішою, ніж шаблонні повідомлення про помилки, які описувалися вище:

```
abstract class EmptyCatchOrFinallyBlock(tryBody: untpd.Tree,
    errNo: ErrorMessageID)(implicit ctx: Context)
  extends Message(EmptyCatchOrFinallyBlockID) {
    val explanation = {
      val tryString = tryBody match {
        case Block(Nil, untpd.EmptyTree) => "{}"
        case _ => tryBody.show
}
```

По суті, Dotty бере блок коду з вихідного коду, зберігає його в проміжних змінних, таких як tryString , а потім додає ці змінні в розширене пояснення. Але ми могли б уявити собі більш складну реалізацію цієї ідеї,

наприклад, маніпулювання частиною пояснення природною мовою на основі контексту коду.

Визнаючи, що розробники часто звертаються до інших джерел інформації, таких як Інтернет, щоб розібратися з помилками чи проблемами налагодження, існує кілька інструментів дослідження для генерації релевантних контексту розширених пояснень. Відомою є реалізація, що виявляє потенційно пояснюваний код на веб-сторінці, аналізує його та генерує на місці пояснення природною мовою та демонструє код. Відповідно це може зменшити потребу в довідковій документації в завданнях модифікації коду. Система HelpMeOut для студентів-початківців збирає приклади змін коду, які виправляють помилки, а потім пропонує ці приклади як рішення іншим. Інструменти Prompter, Seahawk і SurfClipse автоматично отримують відповідні пояснення переповнення стека та представляють їх у IDE. Доступність цих інструментів також свідчить про те, що розробники вважають корисними пояснення, створені людьми, наприклад щодо переповнення стека (Stack Overflow).

2.4 Виведення та інтерпретація помилок як Type Errors

Замість того, щоб збирати повідомлення про помилки з каталогу, інструменти програмного аналізу можуть використовувати обчислювальні методи для автоматичної побудови пояснень у повідомленнях про помилки.

У цьому пункті опишемо системи типів, як одну форм інструменту аналізу програм для автоматичної побудови пояснень. Щоб пояснити, як працюють системи типів, розглянемо кілька простих виразів Haskell, перед якими стоїть команда :t , яка повідомляє нам тип виразу:

```
:t True
:t 'a'
```

Як і очікувалося :t True є True::Bool є логічним значенням true або false. Так само результатом :t 'a' є 'a' :: Char або символ.

Розглянемо функцію під назвою map: яка повертає новий список, беручи список і застосовуючи функцію до кожного елемента в цьому списку. Ось канонічна реалізація map:

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

Haskell може повідомити нам тип map за допомогою:

```
map :: (t -> a) -> [t] -> [a]
```

Сигнатура типу говорить нам, що map приймає функцію, яка приймає t і повертає a , а також список t, а потім він повертає список. Зауважимо, що у наведених вище прикладах не було вказано тип виразів явно, хоча це можна легко реалізувати:

```
:t True :: Bool
:t 'a' :: Char

map :: (t -> a) -> [t] -> [a]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Натомість Haskell може зробити висновок про те, що тип є символьним або логічним за допомогою визначення типу: процесу реконструкції відсутньої інформації про тип через те, як вона використовується в програмі. Під час аналізу програми перевірка типів перевіряє правильність типів. Якщо виявлено порушення, це призводить до повідомлення про помилку типу. Давайте спричинимо помилку типу через наступний фрагмент Haskell:

```
True && 1
```

Оскільки поєднання логічного значення з числом несумісне в Haskell, це очікувано повертає повідомлення про помилку у формі помилки типу:

```
<interactive>:25:9: error:
    • No instance for (Num Bool) arising from the literal ‘1’
    • In the second argument of ‘(&&)’, namely ‘1’
      In the expression: True && 1
      In an equation for ‘it’: it = True && 1
```

Навіть цей простий приклад ілюструє правило з двома кінцями щодо повідомлень про помилку типу. Для системи, на відміну від повідомлень, створених людьми, засіб перевірки типів поглинає основну частину роботи, проходячи стандартний висновок типу, а аналіз програми може механічно створити проблему. Однак для розробника такий стиль виведення означає, що він часто повинен мати точне розуміння алгоритму виведення типу, щоб зрозуміти повідомлення про помилку. Гірше того, покладаючись на висновок типу як на єдиний механізм для індукції повідомлень про помилки, можна дійти до не інтуїтивних повідомлень про помилки навіть для простих проблем. Розглянемо такий приклад:

```
print 5
```

Ця програма, звичайно, друкує 5. А як щодо наступної?

```
print -5
```

Якщо користувач не досвідчений розробник Haskell, то він може бути здивований, виявивши, що це призводить до такого повідомлення про помилку:

```
<interactive>:26:1: error:
    • Non type-variable argument in the constraint: Num (a -> IO ())
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a. (Show a, Num (a -> IO ())) => a -> IO ()
```

Ця помилка Haskell є насправді правильна, але опис цього повідомлення про помилку в термінах системи типів робить його майже неможливим для розуміння. Проблема стає очевидною, лише якщо

користувач усвідомлює, що ' - ' сам по собі є функцією, яка маскується під унарне заперечення: print (- 5)

Отже, виправлення полягає в написанні:

```
print (- 5)
```

Повідомлення про помилки, подібні до наведених вище, властиві мовам програмування, які покладаються на системи типів як засіб, за допомогою якого подаються помилки. Проблеми незрозумілих помилок типу добре відомі, і вплив цих повідомлень про помилки невизначеного типу є вже сформульованими:

- По-перше, помилки невизначеного типу є основною перешкодою для вивчення функціональних мов і вимагають від розробника розуміння базового алгоритму висновку, який використовується.
- По-друге, він зазначає, що є ймовірність повідомляти про всі повідомлення про помилки через механізм виведення типу, але ця стратегія може швидко призвести до багатослівних і незрозумілих повідомлень.
- По-третє, навіть якби аналіз програми міг видати коротке повідомлення про помилку, засіб перевірки типів все одно повинен був би прийняти рішення про те, яке з багатьох можливих місць насправді повідомити.
- По-четверте, алгоритми виведення типу пов’язані з добре відомими твердженнями, такими як вліво-вправо, у яких засіб перевірки типів систематично повідомляє про конфлікти типів ближче до кінця програми. Таким чином, місце, в якій не вдається вивести тип, може не бути місцем, в якій розробник зробив помилку.

Є кілька підходів до дослідження, які роблять помилки типу більш корисними для розробників. Одним із підходів є вибіркова заміна або доповнення автоматизованої діагностики виведення типу повідомленнями про помилки, створеними людиною, для типових ситуацій. Helium, зручний компілятор для вивчення Haskell, використовує цей підхід. Він застосовує

широкий діапазон евристик, щоб представити створені людиною підказки, які доповнюють механізм виведення типів. Подібним чином підхід, який використовує Objective Caml, «виправляє» компілятор: компілятор розглядає перше визначення верхнього рівня, яке не пройшло перевірку типу і намагається зіставити помилку з набором повністю створених вторинних евристик. Ці евристики визначають чи може інструмент представити альтернативне повідомлення про помилку, створене людиною. Якщо так, представлено альтернативне повідомлення; інакше розробнику буде надано типову помилку типу. Їх реалізація охоплює загальновідомі функції мови програмування Objective Caml. По суті, Objective Caml представляє раціональну реконструкцію вихідної помилки типу.

Щоб покращити типи повідомлень про помилки, існують дослідження де застосували статистичні методи, машинне навчання та підходи на основі пошуку. Також застосовують байєсівські міркування до систем типів, оскільки байєсівська модель включає знання про типові помилки розробників, щоб точніше визначити місце справжньої помилки. У випадку застосування машинного навчання використовують помилку типу від компілятора та пов’язані з ним функції вихідного коду, щоб визначити точніше місце порушення проблеми та запропонувати виправлення, які б усунули неправильно введену проблему. Є відомим підхід, за якого перевірка типів сама по собі не створює остаточне повідомлення про помилку. Замість цього засіб перевірки типів використовується як оракул для процедури пошуку, яка знаходить подібні програми, які перевіряють тип.

Інші підходи застосовують формальні методи для покращення повідомлень про помилки. Так використовується орієнтований на виправлення підхід до повідомлень про помилки, заснований на ізоморфізмі за модулем, коли висновок типу не вдається. Отримане повідомлення про помилку потім представляється у формі запропонованої зміни. Наприклад, у списку джерел:

```
val oneToThreeStrings = map ([1, 2, 3], Int.toString) ;
```

компілятор ML зазвичай повідомляє:

```
! Toplevel input:
! val oneToThreeStrings = map ([1, 2, 3], Int.toString) ;
!                          ^^^
! Type clash: expression of type
!   ('a -> 'b) -> 'a list -> 'b list
! cannot have type
!   'c * 'd -> 'a list -> 'b list
```

Натомість McAdam повідомляє таке повідомлення про помилку:

```
Try changing
  map ([1, 2, 3], Int.toString)
To
  map Int.toString [1, 2, 3]
```

На рисунку 2.1 лістинг вихідного коду перетворюється на абстракцію вихідного коду, придатну для перевірки засобом перевірки моделі, разом із специфікацією, що описує певну властивість того, як має поводитися програмне забезпечення. Враховуючи ці вхідні дані, засіб перевірки моделі може виявити порушення властивості. Якщо порушення існує, засіб перевірки моделі повертає контрприклад того, як властивість може бути порушено.

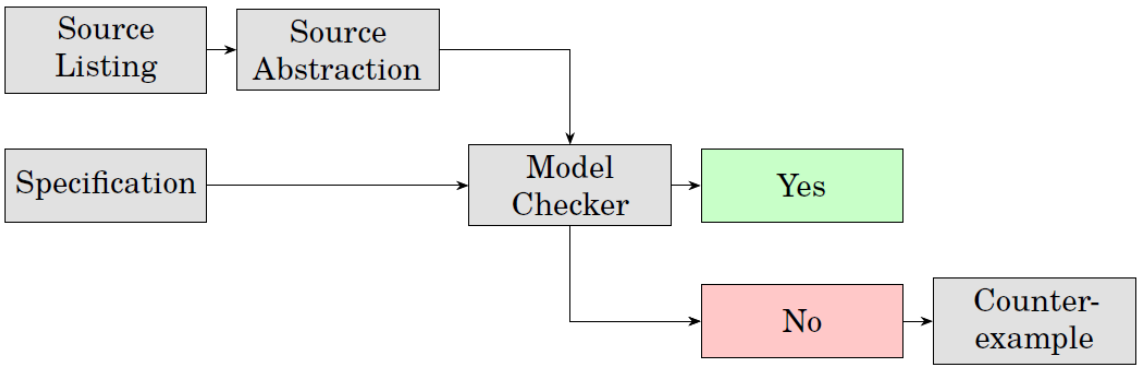


Рисунок 2.1 – Типова модель конвеєра перевірки

Додаткові формальні методологічні підходи до покращення повідомлень про помилки типу включають адаптацію та використання процедур обмеження типу під час аналізу програми для сприяння побудові повідомлень про помилку, використання обґрунтування потоку даних для

пояснення помилок типу, включення різних модифікацій базового виведення типу алгоритм для забезпечення запису конкретних кроків міркування, що використовуються під час логічного висновку, уможливлення інтерактивних підходів до типів запитів та впровадження алгоритмічних методів для точного визначення фактичного місця проблеми.

2.5 Перевірка моделі програмного забезпечення Counterexamples методом

Перевірка моделі програмного забезпечення є іншою формою автоматизованого підходу для створення повідомлень про помилки, яка полягає в формальній перевірці деяких властивостей програмного забезпечення. Принцип перевірки моделі полягає в тому, що, враховуючи кінцевий граф переходів станів системи (тобто модель програмного забезпечення) і формальну специфікацію, перевірка моделі систематично перевіряє, чи виконується специфікація в графі переходів станів. По суті, засіб перевірки моделі досліджує цей кінцевий граф переходів між станами. Якщо засіб перевірки моделі не виявить порушення властивості, то це означає, що модель задовольняє специфікації. В іншому випадку засіб перевірки моделі повідомляє про діагностичний контрприклад або трасування, що порушує специфікацію. Ці діагностичні контрприклади можуть бути представлені окремо або як компонент пояснення повідомлення про помилку.

Існує безліч інструментів перевірки моделей для області розробки програмного забезпечення, включаючи SLAM, SPIN, BLAST і Java PathFinder.

Наведемо приклад перевірки моделі за допомогою CBMC — засобу перевірки моделі для програм C і C++, що використовується до діаграми на рисунку 2.1. Для цього прикладу використаємо наступний код, написаний мовою C:

```
1  void f(int a, int b, int c) {
2    int temp;
3    if (a > b) {temp = a; a = b; b = temp;}
4    if (b > c) {temp = b; b = c; c = temp;}
5    if (a < b) {temp = a; a = b; b = temp;}
6
7    assert (a <= b && b <= c);
8  }
```

Призначення цієї функції полягає в тому, що після виконання операторів if у рядках 3–5, значення змінних буде поміняно місцями так, що a <= b і b <= c у рядку 7. Ця специфікація вбудована за допомогою оператора assert (рядок 7).

Чи відповідає наша програма цій специфікації? Оскільки ця програма є штучно тривіальною, то можна вручну ідентифікувати контрприклад і перевірити, чи він порушує специфікацію. Наприклад, якщо a = 1 , b = 1 і c = 0, то результуючі значення будуть a = 1 , b = 0 і c = 1 у рядку 7. Звичайно, інструмент аналізу програми CBMC також виявляє, що вихідний список порушує специфікацію та видає наступне:

```
State 17 file file.c line 1 thread 0
------------------------------------------------


  INPUT a: 124955 (00000000000000011110100000011011)

State 19 file file.c line 1 thread 0
------------------------------------------------
  INPUT b: 256027 (00000000000000111110100000011011)

State 21 file file.c line 1 thread 0
------------------------------------------------

  INPUT c: 124954 (00000000000000011110100000011010)

Violated property:
  file file.c line 7 function f
  assertion a <= b && b <= c
  a <= b && b <= c


** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```

Засіб перевірки моделей корисний, оскільки створює конкретні приклади як докази існуючої проблеми. Однак контрприклади, які вони створюють, як показано у випадку CBMC, є довільними. Як і у випадку з помилками в попередньому пункті, представлений контрприклад відображає те, що було знайдено під час виконання внутрішнього алгоритму, а не те, що обов'язково є найкращим прикладом для представлення розробнику.

Це призводить до принципових підходів до представлення результатів у інструментах перевірки моделі. Наприклад, такі інструменти як Aluminum і Razor надають користувачеві мінімальний прикла, який містить не більше інформації, ніж необхідно. Існують методи оцінки результатів перевірки моделі через дослідження користувачів, наприклад, вони оцінюють, чи допомагає мінімізація контрприкладів користувачам.

Відомою є застосування ідеї причинності для формального визначення набору причин невдачі специфікації на заданому сліді контрприкладу; ці причини позначені червоними крапками та представлені користувачеві як візуальне пояснення несправності. Amalgam дозволяє розробнику запитувати засіб перевірки моделі про контрприклади, які він надав, запитаннями типу «чому?», "чому ні?".

Деякі підходи використовують кілька контрприкладів для полегшення розуміння або доповнюють контрприклади додатковою інформацією. Стверджується, що, хоча перевірка моделі є ефективною для пошуку елементарних помилок, хоча важчі помилки може бути важко зрозуміти лише на одному контрприкладі. Тому пропонується автоматизована техніка для пошуку кількох контрприкладів помилки, а також причин, які не викликають помилки. Далі виконується аналіз цих виконань, щоб створити більш стислий опис ключових елементів помилки.

Подібним чином представляється алгоритм, який використовує наявність правильних трасувань, щоб локалізувати причину помилки в тілі помилки, повідомляти про одну трасу помилки для кожної причини помилки та генерувати кілька трасувань помилок, що мають незалежні причини. Також

існує техніка, за допомогою якої розробники можуть збільшити масштаб потенційних дефектів програмного забезпечення, аналізуючи один конкретний контрприклад. Для цього коментують контрприклади додатковими етапам, що дозволяє користувачеві краще зрозуміти контрприклади.

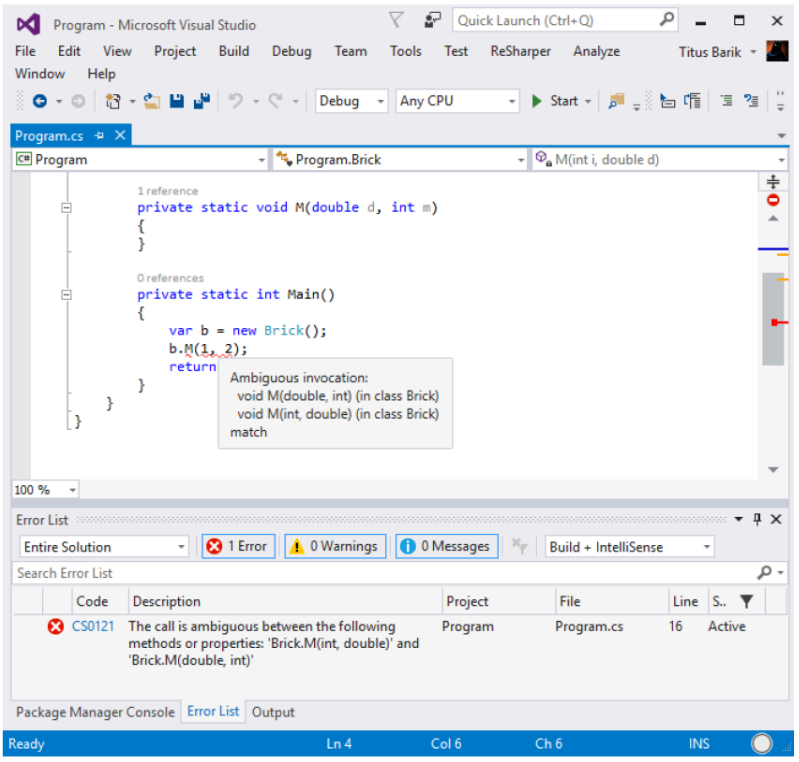
Ще є відомою є концепція перевірки моделі, подібна до покриття коду, в якій пропонується метрика покриття для оцінки повноти набору властивостей після перевірки моделі.

2.6 Візуальне представлення програмного аналізу

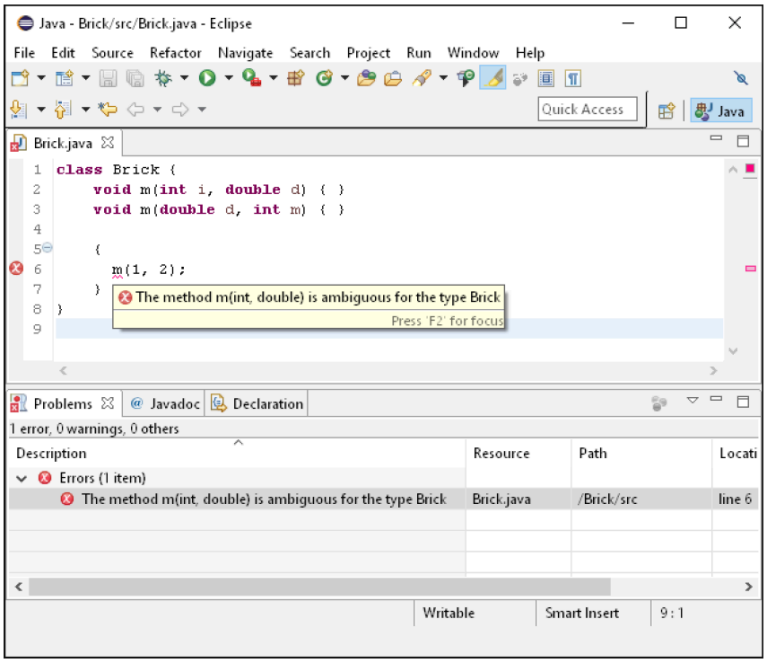
Досі ми представляли повідомлення про помилки переважно через текстові інтерфейси терміналу. Крім терміналів, багато розробників програмного забезпечення використовують сучасні інтегровані середовища розробки (IDE), такі як Eclipse, Visual Studio і IntelliJ, як основну частину свого процесу розробки. Ці середовища призначені для підвищення продуктивності розробників шляхом об’єднання кількох інструментів і надання цих інструментів доступності в рамках єдиного розробницького досвіду.

Не дивно, що повідомлення про помилки від інструментів програмного аналізу також стають доступними в цих середовищах: діагностичні об’єкти роз’єднуються та відображаються через відповідні елементи в IDE.

Розглянемо Visual Studio IDE на рисунку 2.2a з файлом проекту, що містить виклик неоднозначного методу. IDE надає розробнику повідомлення про помилки через кілька елементів інтерфейсу. Вікно списку помилок у нижній частині IDE відображає поточні повідомлення про помилки в проекті. У цьому вікні помилки можна шукати, сортувати та фільтрувати. Компоненти об’єкта діагностики, такі як опис, файл і номер рядка, представлені окремо в таблиці.



а) Visual Studio



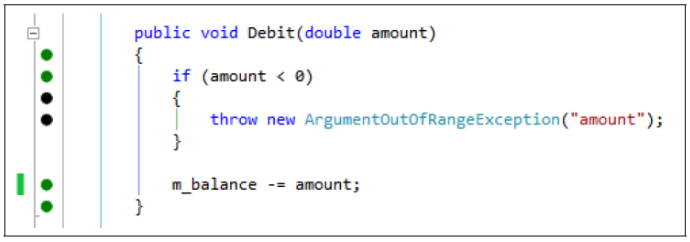
б) Eclipse

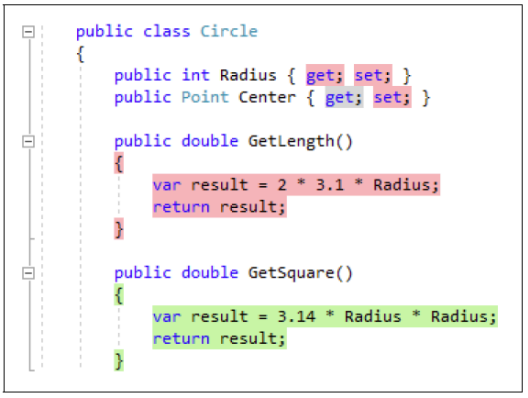
Рисунок 2.2 – Представлення помилок в сучасних IDE

У редакторі вихідного коду для кожного повідомлення про помилку під відповідним вихідним кодом відображається червона хвиляста лінія.

Розробник може навести курсор на хвилясте підкреслення, щоб відкрити спливаючу підказку з описом повідомлення про помилку. Крім того, поле містить візуальні індикатори, які забезпечують огляд того, де проблеми знаходяться в поточному файлі. Розширення модульного тестування, такі як NCrunch і dotCover, використовують індикатори і підсвічування вихідного коду, щоб вказати охоплення модульного тестування (рис. 2.3).

На рисунку 2.3 показано це інструменти для паралельного модульного тестування та покриття коду, які інтегруються з Visual Studio. Тут показано два способи відображення інформації про покриття коду: а) у NCrunch, як маркери підсвічування на полях і б) у JetBrains dotCover, як виділення кольором самого коду. Зелений означає, що тести пройшли, червоний означає, що принаймні один тест, який є тест кейсі, не пройшов, а чорний або сірий показує непокритий код.





а) метод підсвічування на полях б) виділення кольором коду з помилкою

Рисунок 2.3 – Інструменти NCrunch і JetBrains dotCover для виконання тестування

Eclipse IDE на рисунку 2.2 б використовує можливості, подібні до Visual Studio, а інші популярні IDE, такі як Atom, Xcode і Sublime Text – майже однаково подають помилки розробникам.

Декілька дослідницьких інструментів досліджують діаграмне представлення повідомлень про помилки у формі прямокутника та стрілки.

Наприклад, інструмент Refactoring Annotations відображає діаграмний потік керування та інформацію про потік даних і накладає цю інформацію на вихідний код; завдяки цим анотаціям розробники розуміють причини помилок рефакторингу значно швидше й точніше, ніж стандартні повідомлення про помилки Eclipse. Дослідження повідомлень про помилки візуального компілятора ґрунтується на рефакторингових анотаціях. MrSpidey розроблено як зручний інтерактивний статичний налагоджувач для Scheme. Даний інструмент допомагає розробникам точно визначити помилки під час виконання та використовує стрілки, накладені на вихідний код, щоб пояснити розробнику частини графіка потоку вартості. Він рендерить стрілки між пов’язаними елементами у файлах вихідного коду і автоматично впорядковує вікна, коли стрілки охоплюють декілька файлів.

Розширення Rust Enhanced для Sublime Text вставляє інформацію про повідомлення про помилку через фантоми. Фантоми схожі на підказки, за винятком того, що інформація вбудована безпосередньо в текстове подання та залишається постійною.

Відображення повідомлень про помилки також представляють інформацію разом із вихідним редактором. Існує реалізація візуалізації помилок компілятора в Java, при якій вноситься візуальний синтаксис, який візуалізує діаграмне представлення для неправильних призначень, перевірки типу та винятків.

Інструмент Stench Blossom — це візуалізація, що складається з півкілець у правій частині панелі редактора; кожен сектор відповідає категорії помилок. Інструмент створено для випадків, коли в коді може бути кілька одночасних проблем, і де виявлені проблеми вимагають досвіду розробника, щоб визначити, чи справді проблему потрібно вирішувати. Дослідження показує, що така візуалізація допомагає розробникам робити більш обґрунтовані судження про код, який вони написали.

Інші повідомлення про помилки розробникам представляються як інформаційні панелі та сповіщення електронною поштою.

2.7 Помилки розробників

Типовий компілятор, наприклад для Java або C#, розпізнає кілька тисяч можливих помилок при аналізі програми. Зважаючи на обмежені ресурси, доцільно охарактеризувати простір повідомлень про помилки, які фактично отримують розробники, щоб ефективно використовувати та вдосконалювати інструменти.

Щоб зрозуміти об'єм проблеми було виконано емпіричне тематичне дослідження на основі 26,6 мільйонів фрагментів коду Java та C++ в Google, які були створені протягом дев'яти місяців тисячами розробників. Було виявлено, що майже 30% коду у Google виходить з ладу через помилку статичного аналізу, а середній час вирішення кожної помилки становить 12 хвилин. Однак дорогі помилки, які допускають розробники, є досить тривіальними і пов'язані з такими проблемами як залежності, невідповідності типів, синтаксичні та семантичні помилки.

Для розробників-початківців, які використовують Java в IDE, ситуація ще гірша. За допомогою телеметрії понад 37 мільйонів подій компіляції виявили, що майже 48% усіх компіляцій зазнають невдачі через помилку компілятора. Подібно до помилок експертів-розробників із Google, новачки також робили синтаксичні та семантичні помилки.

Використовуючи Python в 1,6 мільйона фрагментів коду, з яких 640 000 призвели до помилки (приблизно 40%), і повторно перевіряючи набір даних було підібрано модель розподілу цих повідомлень про помилки та виявивлено, що вони емпірично схожі на розподіл Ципфа-Мандельброта. Такі степеневі розподіли мають невеликий набір значень, які домінують у розподілі, за якими йде довгий хвіст, який швидко зменшується.

Інші дослідники також виявили степеневий розподіл для повідомлень про помилки. Спочатку було емпірично отримано та каталогізовано діагностику помилок COBOL для визначення розподілу Парето: 20% відсотків типів помилок становили 80% загальної частоти помилок. Потім

було встановлено, що перші десять помилок становлять майже 52% усіх помилок, а двадцятка найбільших – 62,5% усіх помилок.

Також виявлено, що 73% усіх поданих документів не вдалося скомпілювати через синтаксичну помилку; у верхньому квартилі майже 50% усіх поданих матеріалів не вдалося скомпілювати через синтаксичну помилку. Середня кількість рядків коду для поданих програм становила вісім.

Незважаючи на те, що не виконувався підгін підгін дистрибутивів за моделлю, візуальний огляд Java та C++ свідчить про наявність подібного степеневого ефекту.

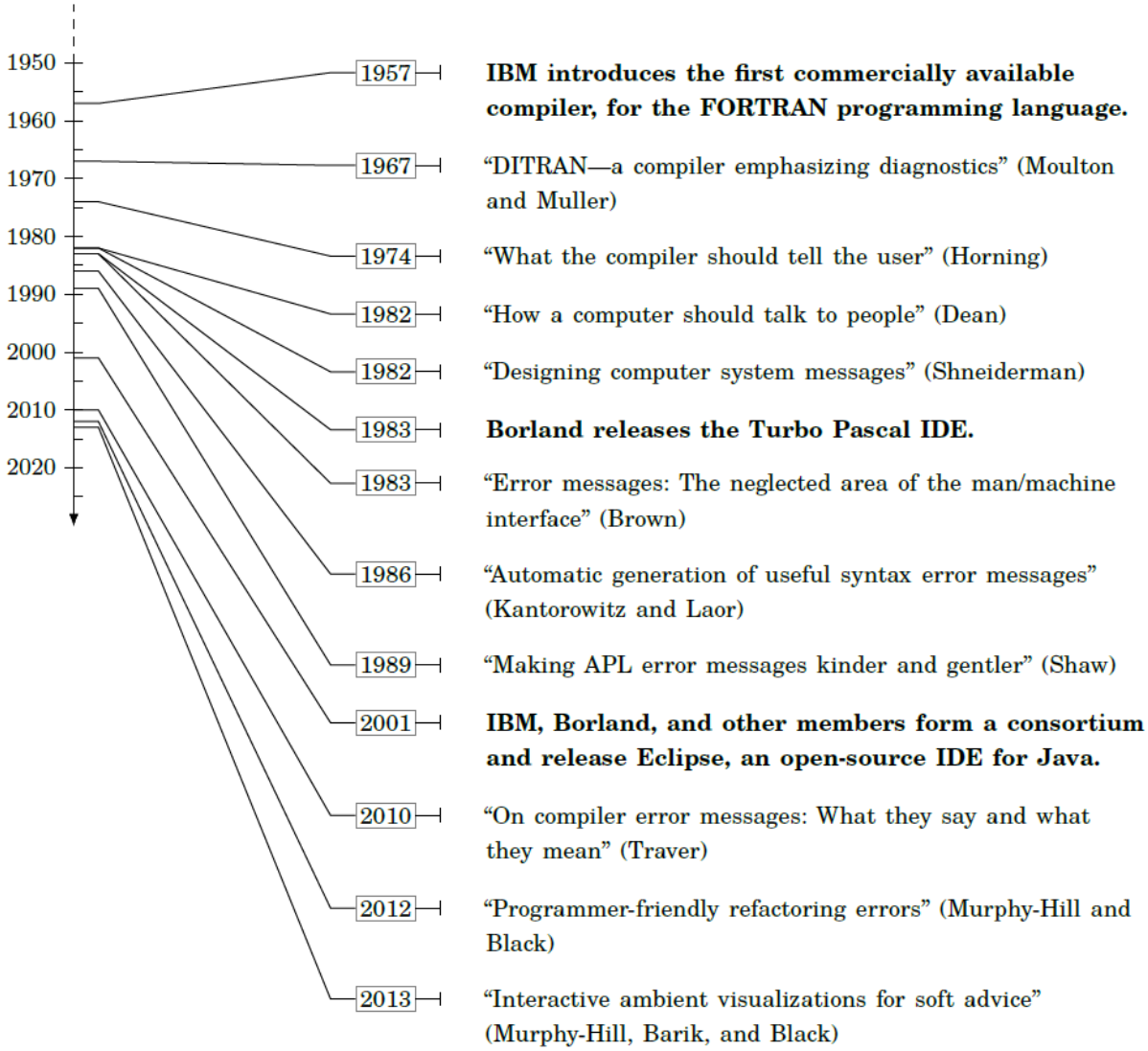


Рисунок 2.4 – Історія щодо дизайну проектування повідомлень про помилки в інструментах програмного аналізу

На рисунку 2.4 у поєднанні з рекомендаціями щодо дизайну значні зміни в галузевих інструментах виділено жирним шрифтом.

Тріангуляція цих кількох досліджень вказує на дві узгоджені особливості щодо помилок у мовах програмування. По-перше, домінуючі помилки, як з точки зору вартості, так і частоти, є відносно послідовними, незалежно від досвіду розробника. Відповідно, вдосконалення цих повідомлень про помилки принесе користь багатьом розробникам. По-друге, степеневий закон розподілу припускає, що усунення навіть невеликої кількості домінуючих помилок зробить інструменти програмного аналізу більш зручними.

2.8 Методології розробки повідомлень про помилки

В даному пункті роботи в описовому вигляді представимо методології, методи і рекомендації з точки зору розробки дизайну повідомлень про помилки.

Відомими є рекомендації, щоб усі повідомлення про помилки представлялися мовою вихідного коду: зокрема, повідомлення про помилки повинні містити назву змінної, яка фактично використовується у вихідному коді, а не адресу пам’яті змінної. Також особлива увага зосереджується на структурі: в ідеалі повідомлення про помилки повинні пропонувати виправлення.

Методика Хорнінга повторює подібні вказівки, а саме, що хороші повідомлення про помилки орієнтовані на джерело. Крім того, читабельні повідомлення про помилки повинні описувати ознаки проблеми. В даній методиці вводяться деякі нові ідеї щодо реляційності, де зазначається, що якщо невідповідність стосується інформації з іншої частини програми, наприклад, оголошення, тоді цю інформацію слід відображати к адресу пам’яті, при чому важливо розглядати взаємодію між розробниками та інструментами у вигляді діалогу.

Цей діалог, від інструментів до розробників, розглядається як замінні виправлення: «очікував це , виявив те». Ці типи повідомлень про помилки є корисними і, безперечно, мають своє місце, але насправді вони не є раціональними реконструкціями. Тим не менше можна розглядати ці рекомендації як первинні елементи для мислення про раціональну реконструкцію. Для розширення цього мислення, вводять низку орієнтованих на людину вказівок, отриманих із різних джерел. Зокрема, для раціональної реконструкції мають відношення вказівки щодо представлення повідомлень розмовною мовою, і вказівки, які припускають, що необхідність перечитування повідомлень про помилки є сигналом того, що вони заплутані.

Згідно методики Шнейдермана додається більше нових рекомендацій настанов («вирішувати проблему в термінах користувача») для раціональної реконструкції; однак ця робота є складною, оскільки вона сприяє контрольованому експерименту для представлення різних повідомлень про помилки. Дослідження виявило, що застосування цих інструкцій щодо дизайну до повідомлень про помилки спонукало учасників успішно вирішувати дефекти частіше, ніж базові повідомлення.

У 1983 році було випущено інтегроване середовище розробки Borland Turbo Pascal для спільноти розробників. Ця IDE представила безліч нових можливостей, включаючи багатовіконні середовища, які дозволяли розробникам взаємодіяти, наприклад, з вікном вихідного коду та вікном виведення помилок одночасно. Відповідаючи на можливості, створені IDE, було визначено, що ці «сучасні» системи також пропонують можливості для значного покращення повідомлень про помилки. Дані рекомендації використовують наступні візуальні можливості: використання кольору для визначення невірних символів у вихідному коді, надання контекстного вікна, яке відображає кілька рядків вихідного коду до та після повідомлення про помилку, а також надання деяких візуальних маркерів, щоб показати розробнику, де у вихідному коді сталася помилка. Окремо ці вказівки не є раціональними реконструкціями; однак легко зрозуміти, як підсвічування та

візуальні маркери можуть дозволити раціональним реконструкціям більш виразно передавати свої пояснення.

Також є відомою концепція де пропонується використовувати візуальні презентації синтаксичних помилок. Наприклад, під час синтаксичної помилки вони використовують ' | ' для позначення першого символу, який не відповідає синтаксису мови, а '.', щоб вказати, де компілятор відновив аналіз, і коротке повідомлення, що пропонує інші маркери для заміни (' , ' або ' ; '):

```
const n=5: b:=4;
         |-----. { , ; } EXPECTED
```

Незважаючи на покращене візуальне представлення, ці повідомлення про помилки все ще мають фундаментально очікувану форму. Лише в рекомендаціях Шоу не використовують дану форми представлення помилок. Зокрема, Шоу вводить суттєвий внесок у раціональну реконструкцію: повідомлення про помилки повинні вказувати причину помилки. Замість повернення (в APL), наприклад:

```
Length Error
       2 3 + 4 5 6
```

компілятор натомість мав би видати:

```
Shape Error - Length of corresponding axes must be equal.
       2 3 + 4 5 6
       2 | 3
       ^   ^
```

Друге повідомлення про помилку пояснює, чому довжина (форма) недійсна. Відповідно, помилка також використовує деякі візуальні представлення, через ' ^ ' і ' | '.

Після появи попередної методики в інструкціях з проектування, важливою є поява графічних інтерфейсів користувача, замість текстових інтерфейсів користувача.

Дві додаткові рекомендації щодо дизайну інформації про помилки згідно методології Мерфі-Хілла та Блека вносять елементи раціональної

реконструкції через представлення візуальних анотацій, таких як прямокутники та стрілки, встановлені безпосередньо на вихідний код, який фактично редагує розробник. Обидві рекомендації щодо проектування містять декілька корисних рекомендацій, але одна з них настанова реляційності найближча до ідей у межах раціональної реконструкції: настанови стверджують, що порушення часто виникають через численні програмні елементи в коді, і що візуалізація повинна виявляти ці зв’язки. Іншими словами, візуалізація повинна представляти причинно-наслідкову форму раціональної реконструкції повідомлень про помилки.

Висновки до розділу 2

В даному розділі представлено огляд інструментів аналізу програм з метою ознайомлення з поточними питаннями про те, як сучасні інструменти аналізу програм створюють і передають повідомлення про помилки розробникам. Повідомлення про помилки розглядаються через теоретичну структуру раціональної реконструкції, наголошуючи виборі дизайну, також помилки представлено в контексті інструментів аналізу програм, які розробники фактично використовують на практиці.

РОЗДІЛ 3. РОЗРОБКА МЕТОДІВ ТА МЕТОДОЛОГІЇ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ІНТЕРПРЕТАЦІЇ ПОМИЛОК В ПРОЦЕСІ ДЕБАГГІНГУ

3.1 Методика інтерпретації та розуміння помилки користувачем

Дослідимо питання, які стосуються того, як розробники сприймають і розуміють повідомлення про помилки різними способами, якими вони представлені в інтегрованих середовищах розробки.

Давайте розглянемо, як повідомлення про помилки сприймають розробники у своїй IDE. Дослідження проведемо через гіпотетичного розробника, Баррі. Баррі нещодавно приєднався до великої компанії, що займається програмним забезпеченням, і йому потрібно реалізувати деякі відсутні функції в бібліотеці структур даних. Будучи відносно новачком у цій бібліотеці, він звертається до свого колеги з проханням допомогти розпочати роботу. Зрештою він отримує відповідь від свого колеги з коротким фрагментом коду.

Баррі копіює та вставляє цей фрагмент у свій редактор і дивується, що IDE видає помилку. Він зосереджує свою увагу, тобто візуально фокусується на тексті помилки на панелі задач у нижній частині екрана (рис. 3.1):

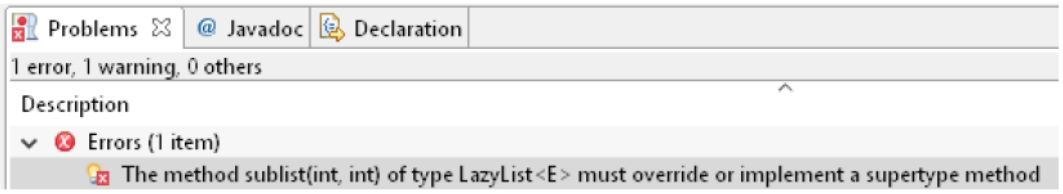


Рисунок 3.1 – Інтерпретація помилки засобами IDE

Він переглядає повідомлення про метод, а потім двічі клацає помилку на панелі задач. Це перенаправляє IDE до редактора вихідних кодів, і Баррі підтверджує, що помилка пов’язана з кодом, який він щойно додав. На полях

редактора вихідного коду він тепер помічає піктограму лампочки, на яку наводить курсор, щоб відкрити спливаюче вікно з помилкою (рис. 3.2):

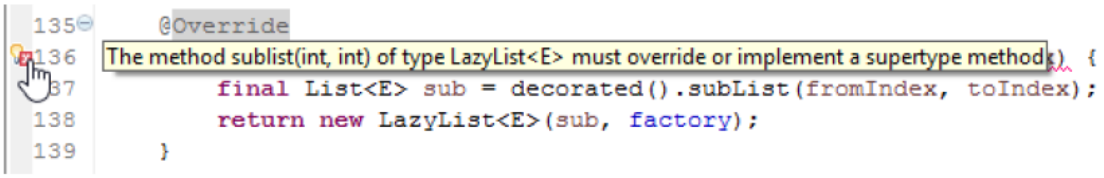


Рисунок 3.2 – Спливаюче вікно помилки

Але, спливаюче вікно є менш корисним, ніж він очікував, оскільки воно повторює повідомлення, яке є на панелі задач. Крім того, спливаюче повідомлення про помилку приховує сигнатуру методу, де, на його думку, насправді знаходиться проблема.

Далі він помічає червоне хвилясте підкреслення, яке в Eclipse вказує на наявність помилки. Баррі наводить курсор на підкреслення, відкриваючи спливаюче вікно Quick Fix. На відміну від спливаючого вікна помилки, спливаюче вікно Quick Fix містить можливі «виправлення», які змінюють вихідний код, окрім повідомлення про помилку. Він витрачає кілька секунд, розглядаючи повідомлення про помилку та порівнюючи його з запропонованими виправленнями. Його погляд на мить відривається від спливаючого вікна, коли його увагу привертає анотація @Override у вихідному коді. Потім він повертається до спливаючого вікна, оскільки четверта опція також посилається на цю анотацію (рис. 3.3):

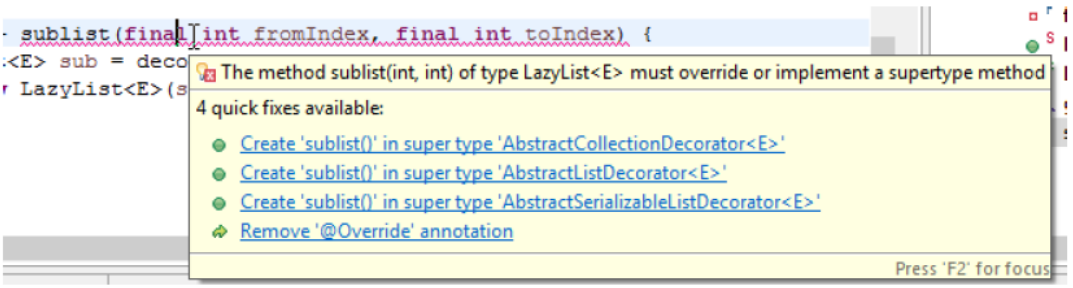


Рисунок 3.3 – Спливаюче вікно Quick Fix

Баррі відомо, що анотація @Override використовується для інформування компілятора про те, що поточний метод має замінити метод у батьківському класі. Щоб перевірити, чи це правда, він переходить до оголошення класу та клацає батьківський клас, утримуючи клавішу Control:

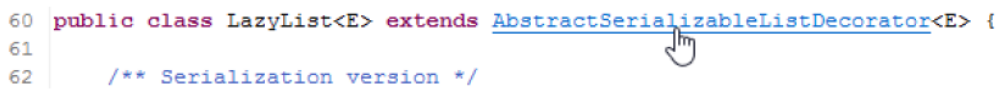


Рисунок 3.4 – Вигляд батьківського класу

Він оглядає структурну панель, яка підсумовує всі методи в класі, і бачить, що батьківський клас містить лише непов'язані методи, такі як writeObject :

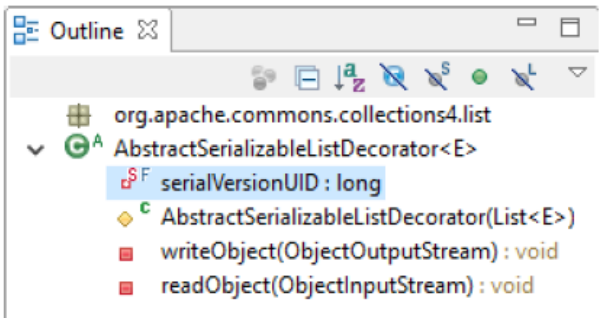


Рисунок 3.5 – Методи класу

Тепер він переконаний, що його колега міг ненавмисно включити анотацію @Override, яка не застосовується до його рішення. Він повертається до початкового класу знову та застосовує виправлення «Видалити анотацію @Override». Eclipse перебудовує проект, і він перевіряє панель проблем, щоб переконатися, що помилки вже немає.

Але 95 % розробників використали саме таку стратегію виправлення помилки як Баррі. На жаль, це виправлення виявилося неправильним. Справжня проблема полягає в тому, що декларація методу sublist написана з помилкою, і її слід було б назвати subList з великої літери L. Баррі міг би

виявити цю орфографічну помилку, якби піднявся ще на одну гілку в ієрархії класів:

```
106    public List<E> subList(final int fromIndex, final int toIndex) {
107        return decorated().subList(fromIndex, toIndex);
108    }
```

Рисунок 3.6 – Вигляд батьківських класів

Однак, цей сценарій не поодинокий для Баррі. Наприклад, публікація з найвищим рейтингом на StackOverflow (http://stackoverflow.com/questions/94361/when-do-you-use-javas-override-annotation-and-why) щодо помилки анотації @Override свідчить про те, що вона переважно виникає в ситуаціях, коли назви методів були написані з помилками.

Чому Баррі не дуже прискіпливо звернув увагу на повідомлення про помилку ? При детальному розгляді повідомлення про помилку насправді згадуються методи супертипу, хоча і не явно за назвою. Або це також може бути так, що повідомлення про помилку спонукає розробників надавати пріоритет певним рішенням для свого коду над іншими.

Для пропонованої методики розглянемо наступні питання та обґрунтування для кожного:

Питання 1. Наскільки ефективно розробники використовують повідомлення про помилки для різних категорій помилок? Дане питання потрібне для того, щоб оцінити репрезентативність експериментальних завдань щодо дорогих повідомлень про помилки, де вартість визначається як частота помилки, помножена на середній час вирішення. Крім того, результати цього питання надають описову статистику, щоб визначити, чи деякі категорії дефектів важче вирішити, ніж інші. Визначення категорій дефектів, які важче вирішити, може допомогти точно визначити, де раціональна реконструкція може бути найбільш вигідною для розробників.

Питання 2. Чи детально читають розробники повідомлення про помилки ? Незважаючи на те, що IDE представляють повідомлення про

помилки, призначені для використання розробниками, питання, наскільки розробники читають ці повідомлення в процесі вирішення, залишається відкритим. Наприклад, розробник може використовувати панель задач не для того, щоб прочитати повідомлення про помилку, а тому, що йому відомо, що подвійне клацання повідомлення про помилку на панелі є зручним способом переходу до місця порушення у вихідному коді. Якби це було справді так, то надання інформації про помилку не дуже б допомогло вирішити проблему.

Task	Error Message[1]	Package	Category	Defect Introduced
T1	`The method sublist(int, int) of type LazyList<E> must override or implement a supertype method`	List	Semantic	Renamed `sublist` to `subList`, breaking existing `@Override` annotation.

T2	`The type CursorableLinkedList<E> must implement the inherited abstract method List<E>.isEmpty()`
	
	`The type NodeCachingLinkedList<E> must implement the inherited abstract method List<E>.isEmpty()`
		List	Semantic	Deleted `isEmpty` method from abstract parent class.

T3	`The import org.apache.commons.collections3 cannot be resolved ( ... repeated 50 times)`	Map	Dependency	Changed version of `collections4` to non-existent `collections3` library in `import` statements.
T4	`The method get() is undefined for the type Queue<E>`	Queue	Dependency	Renamed method invocation from `element()` to non-existent `get()`.
T5	`The method add(E) in the type Collection<E> is not applicable for the arguments (int, capture#8-of ? extends E)`	Set	Type mismatch	Added additional argument of `0` to `add` method call.

T6	`Type mismatch: cannot convert from Set<Map.Entry<K,V>> to Set<Map.Entry<V,K>>`
	
	`Type mismatch: cannot convert from Set<Map.Entry<V,K>> to Set<Map.Entry<K,V>>`
		Map	Type mismatch	Swapped key and value in dictionary from `Entry<K,V>` to `Entry<V,K>`.

T7	`Unhandled exception type InstantiationException`	Map	Other	Changed less specific exception `Exception` to `IllegalAccessException`, which is not thrown by the code.
T8	`Duplicate method next() in type EntrySetMapIterator<K,V>` `Duplicate method next() in type EntrySetMapIterator<K,V>`	Iterators	Other	Copied and pasted `next` method to create duplicate method.
T9	`Cannot make a static reference to the non-static type E`	Queue	Semantic	Added `static` modifier to `readObject` method.
T10	`Syntax error on token "default", : expected after this token`	Map	Syntax	Removed `:` from `default:` in `switch` statement.

Рисунок 3.7 – Завдання помилок компілятора

Питання 3. Чи важко вирішити помилки компілятора через повідомлення про помилку ? Усунення помилок компілятора в IDE вимагає від розробників виконання комбінації дій, таких як навігація до файлів і редагування вихідного коду. Одна з гіпотез полягає в тому, що певні помилки компілятора важко вирішити не тому, що саме повідомлення про помилку є не дуже зрозумілим, а тому, що завдання вимагає складних модифікацій коду, щоб усунути дефект. Крім того, може статися так, що рішення потребує лише простої зміни коду для виправлення дефекту, але незрозуміле повідомлення

про помилку заважає розробнику виявити необхідну зміну коду. Таким чином, потрібно зрозуміти, наскільки неякісні повідомлення про помилки шкідливі для розробника. Якщо складність усунення помилок компілятора пов’язана з повідомленням про помилку, то є вагоме виправдання для проведення досліджень.

3.2 Методологія візуалізації повідомлень про помилки компілятора

Інтегровані середовища розробки, такі як Eclipse, IntelliJ і Visual Studio, пропонують низку візуалізацій, які допомагають розробникам ефективніше ідентифікувати та розуміти повідомлення про помилки під час аналізу програми. Наприклад, як доповнення до текстового опису помилки, знайденої у вихідних даних консолі або спеціальному вікні помилки, ці повідомлення можуть містити індикатор на одному або кількох полях разом із червоним хвилястим підкресленням, накладеним на вихідний текст, що означає, що такого типу візуалізації схематично вказують відповідне розташування помилки. Але використання діаграм значно більше сприяє ефекту самопояснення, ніж текст сам по собі.

Незважаючи на візуальні можливості в сучасних IDE, вважається, що однією з причин, чому візуальні повідомлення про помилки залишаються незрозумілими для розробників, полягає в тому, що вони є недостатньо раціональними. Зокрема, ці візуалізації не підтримують розробників під час самопояснення — по суті процесу, за допомогою якого користувачі самостійно генерують пояснення собі та іншим, щоб зрозуміти ситуацію. Помилки під час процесу самопояснення можуть призвести до суттєвої втрати продуктивності, оскільки недостатньо обізнані і мають мало досвіду.

Стверджується, що розробники отримають перевагу, коли аналіз програми розкриє її автоматизоване міркування за допомогою трасування пояснень та форм конструкцій, у яких повідомлення про помилки зображається ланцюжком у зворотному напрямку чим встановлюється всі

етапи причин помилки. Відомо, що інструменти програмного аналізу можуть використовувати ці структури трасування-пояснення для створення виразних, пояснювальних візуалізацій, які узгоджуються з тим, як розробники самостійно пояснюють повідомлення про помилки. Тому можна стверджувати наступне:

- Базовий набір складних візуальних анотацій допомагають розробникам краще розуміти повідомлення про помилки.
- Оцінка завдання з використанням набору макетів, яка демонструє, що пояснювальні візуалізації дають більш коректні самостійні пояснення, ніж базові візуалізації, які сьогодні використовуються в IDE.
- Оцінка завдання відкликання, під час якої розробники пишуть програми в мінімалістичному середовищі програмування, щоб спеціально генерувати помилки компілятора, що демонструє, що кращі самопояснення дозволяють розробникам створювати кращі описові моделі повідомлень про помилки.

Розглянемо деякий випадок. Андрій — досвідчений розробник C++, який нещодавно перейшов до проекту, який розробляється мовою програмування Java. Під час програмування він стикається з хвилястим червоним підкресленням, як показано на малюнку рис. 3.6 a, що вказує на помилку. Здається, проблема пов’язана з final int i, який Андрій визнає приблизно схожим на концепцію змінної const у C++. Андрій продовжує вивчення помилки та помічає повний текст помилки на нижній панелі свого IDE (рис. 3.8 c).

На рисунку 3.8 представлено порівняння потенційно неініціалізованої помилки компілятора змінної за допомогою:

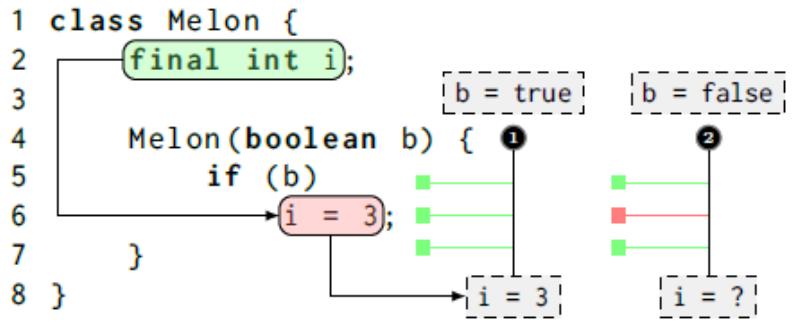
a) базових візуалізацій, програмної парадигми, яка зараз використовується в IDE;

b) пояснювальних візуалізацій;

c) текстового повідомлення про помилку.

```
1    class Melon {
2        final int i;
3
4        Melon(boolean b) {
5            if (b)
6                i = 3;
7        }
8    }
```

(a) Baseline visualization



(b) Explanatory visualization

```
Melon.java:7: error:
   variable i might not have been initialized
    }
    ^
1 error
```

(c) Error message text

Рисунок 3.8 – Приклади подання помилок

Повідомлення про помилку вказує на те, що змінна може не бути ініціалізована в рядку 7. Андрій вирішує, що це повідомлення про помилку є неправильним, і ігнорує його, оскільки рядок 7 містить лише фігурну дужку, яка, не має нічого спільного з його проблемою, оскільки часто в C++ він отримує схожі повідомлення про помилки.

Андрій пояснює це тим, що проблема полягає в тому, що кінцеві змінні в Java, як і константні змінні в C++, повинні бути призначені в точці їх оголошення або в списку ініціалізаторів конструктора. Відповідно він

переписує рядок 2, щоб прочитати final int i = 3, що в свою чергу призводить до помилки нижнього потоку, оскільки рядок 6 тепер відображає неможливість призначити значення кінцевій змінній i. Андрій вважає, що константу не можна перепризначити, тому він видаляє весь умовний оператор. Незважаючи на те, що програма зараз компілюється, виправлення виявляється неправильним.

Проблема тут полягає в тому, що Андрій вивчив евристику того, як змінні працюють в мовах програмування, але ця евристика в цьому випадку не працює. Як і C++, Андрій правий у тому, що кінцеві змінні Java можна призначити лише один раз. Але на відміну від C++, кінцеві змінні в Java можуть бути призначені в точці, відмінній від оголошення.

Ця помилкова гіпотеза залишається невиправленою IDE. В IDE візуалізація червоним хвилястим підкресленням може вказувати лише на одне місце, пов’язане з помилкою. IDE не може передати, що проблема залежить від кількох елементів програми.

Наприклад, текст помилки та вказане розташування є точними, оскільки після цього рядка змінна може бути неініціалізованою, але IDE не має ефективного способу вказати, як це розташування пов’язане з кінцевою змінною.

На відміну від цього, розглянемо підхід, показаний на рисунку 3.8 b. Тут IDE надає візуальне пояснення проблеми у вихідному коді Андрія. Хоча він може ще раз неправильно припустити, що остаточні змінні повинні бути призначені під час оголошення, візуалізація означає, що проблема насправді пов’язана з потоком керування.

Зокрема, пояснювальна візуалізація показує Андрію, що існує кодовий шлях, у якому i присвоюється значення (коли b = true), і інший шлях, де його немає (коли b = false). Цього разу розробник правильно виправляє дефект, додаючи оператор else до умови, ініціалізуючи його відповідним значенням у випадку, коли b = false.

Annotation	Frequency	Description
Point	49	A particular token or set of tokens has been marked. Examples include underlining or circles the token(s).
Text	45	Natural language text. For example, “assign a value to the variable” or “dead code”.
Association	33	An association between two or more program elements, which is accomplished by drawing a connecting line between the elements, with or without arrow heads.
Symbol	20	Symbols include visual annotation such as `?` or `x`, or numbered circles, to name a few.
Code	14	Explanatory code that is written in order to explain the error message, for example, `if (b == false)` or `m(1.0, 2)`. This does not have to be correct Java code, but should be interpretable as pseudocode.
Strikethrough	5	The strikethrough is separated from the point annotation because this annotation is provided by IDEs today, and has pre-established semantics.
Multicolor	-	The use of more than a single color to explain a concept. For example, green may be used to indicate lines that are okay, and red to indicate lines that are problematic. This option was not available to students in the pilot study.

Рисунок 3.9 – Частота візуальних анотацій у Pilot

Отже, цей гіпотетичний сценарій ілюструє, чому програмна парадигма візуалізації недостатня для підтримки процесу самопояснення. Цей сценарій є ілюстрацією більш загальної проблеми з виходом інструментів програмного аналізу: ці інструменти представляють лише кінцевий результат складного процесу міркування.

Розглянемо наступне питання дослідження: Які анотації використовують розробники, коли пояснюють один одному повідомлення про помилки?

Вважається, що якщо розробники віддадуть перевагу певним типам анотацій під час пояснення повідомлень про помилки один одному, вони також зможуть отримати позитивний результат, якщо анотації використовуватимуться для пояснення повідомлень про помилки через IDE.

Згідно експерименту розробники об’єднані в пари для виконання вправи «пояснювач-слухач». Це вправа, у якій одного розробника,

пояснювача, просять усно пояснити повідомлення про помилку іншому, візуально коментуючи перелік вихідного коду під час пояснення. Потім після пояснення ролі міняються, і другий пояснювач прокоментував друге повідомлення про помилку. Ці приклади наведені на рисунку 3.10.

Task Name	OpenJDK File	Error Message
Melon	`VarMightNotHaveBeenInitialized.java`	`variable i might not have been initialized`
Kite	`UnreportedExceptionDefaultConstructor.java`	`unreported exception Exception in default constructor`
Brick	`RefAmbiguous.java`	`reference to m is ambiguous, both method m(int,double) in Brick and method m(double,int) in Brick match`

Zebra	`InferredDoNotConformToBounds.java`	`cannot infer type arguments for BlackStripe<>;`
		`reason: inferred type does not conform to declared bound(s)`
		
		`inferred: String`
		`bound(s): Number`

Apple	`RepeatedModifier.java`	`repeated modifier`
Trumpet	`UnreachableCatch1.java`	`unreachable catch clause` `thrown types FileNotFoundException, EOFException have already been caught`

Рисунок 3.10 – Завдання для пояснень

З цих анотацій було створено таксономію візуальних анотацій на основі спостережень і класифікацію відповіді, використовуючи цю таксономію. Зведені результати наведено в таблиці 3.9.

Дане дослідження сформулювало пояснювальні візуалізації, які можна реалізувати за допомогою анотацій, таких як точки, асоціації, символи та пояснювальний код. Вважається, що ці типи анотацій без є інтуїтивно зрозумілими для використання під час пояснення.

Пропонується набір із восьми візуальних анотацій, які подано на рисунку 3.11. Тепер можна ми описати ці анотації, використовуючи приклад із рисунка 3.6 b. Початкова точка для візуального пояснення в списку вихідного коду вказується за допомогою коду (зелений прямокутник із

закругленими кутами, який оточує елемент програми). У наших макетах візуалізації вибираємо початкову точку такою ж, що й джерело помилки, визначене IntelliJ (рис. 3.1 а). У прикладі це final int i .

Symbol	Description
code	The starting location of the error.
code	Indicates issues related to the error.
	Arrows can be followed. They indicate the next relevant location to check.
❶	Enumerations are used to number items of potential interest, especially when the information doesn't fit within the source code.
?	The compiler expected an associated item, but cannot find it.
×	Conflict between items.
code	Explanatory code or code generated internally by the compiler. The code is not in the original source.
	Indicates code coverage. Green lines indicate successfully executed code. Red lines indicate failed or skipped lines.

Рисунок 3.11 – Візуальні анотації

Продовжуючи приклад, початкова точка пов'язана з другою точкою, int i , тому що саме тут відбувається потенційне призначення змінній. Початкову точку вказуємо кодом (червоний прямокутник із заокругленими кутами), а асоціацію позначаємо (стрілка напрямку).

Друга асоціація веде розробника до блоку пояснювального коду, який містить копію оператора. Пояснювальний код представлений кодом (пунктирний сірий прямокутник), що вказує на те, що оточені елементи є пояснювальними та не є частиною оригінального вихідного коду програми. Цей блок пояснювального коду є частиною більшої складеної анотації, що описує сценарій потоку керування, за яким виконується оператор.

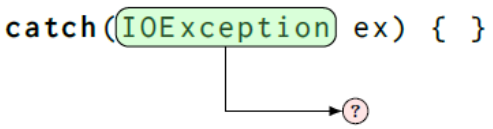
Ця комбінована анотація демонструє, що кілька базових анотацій можна поєднати, щоб створити нову анотацію для вираження більш складної концепції. Одним із цих компонентів є анотація покриття коду. Ця анотація використовує зелена і червона лініяї, щоб вказати, чи покрита лінія. Крім того, переліки 1 і 2 надають розробнику зручні мітки для посилань на гілки (наприклад, «Схоже, що це добре працює в гілці 1, але не в гілці 2»). Останнім компонентом є ще один блок пояснювального коду, що вказує одну можливу умову, за якої буде виконано розгалуження.

Таким чином, складена анотація вказує, що i = 3 , і всі оператори в межах гілки 1 будуть виконані, коли b = true. Потім ця комбінована анотація використовується, щоб показати розробнику контрприклад, у якому і буде неініціалізовано. Просте текстове пояснення про те, що і не ініціалізується, коли b = false , дало б той самий висновок, але ми припускаємо, що проміжні кроки в поясненні важливі для розуміння розробником.

Є дві візуальні анотації, які не з’являються в прикладі, які потребують пояснення. Це червоний хрестик, який вказує на наявність конфлікту між блоками, наприклад, коли розробник випадково вказує повторювані модифікатори:

```
class Apple {
    public public String toString() {
        return "Red";
    }
}
```

Знак питання в червоному кружечку використовується для вказівки на те, що елемент програми повинен бути пов’язаний з іншим елементом, але з’єднувальний елемент не знайдено. Це може статися, у випадку коли оператор catch недоступний, виняток ніколи не може бути викинуто або він завжди перехоплюється попередньою пропозицією catch:



У реальних базах коду розробникам доводиться пояснювати повідомлення про помилки у функціональному коді, переплетеному з помилковим кодом, і в кількох вихідних файлах. Розглянуті завдання містили лише код, безпосередньо пов’язаний із генеруванням помилки, і в одному вихідному файлі. Не повністю досліджено питання чи пояснювальні візуалізації будуть однаково корисними чи масштабуються до більш реалістичних контекстів.

3.3 Методологія інтерпретації помилок за допомогою компілятора

Розглянемо методологію обґрунтування-пояснення раціональної реконструкції, модель аргументу Тулміна, щодо розробки та оцінки повідомлень про помилки компілятора. Визначимо повідомлення про помилки, які реалізують модель Тулміна за структурою та змістом, як пояснювальні повідомлення про помилки.

Розглянемо приклад поганих пояснень повідомлень про помилки, на основі наступного фрагменту коду Java:

```
2  void m() {
3    final int x;
4    while (true) {
5      x = read();
6    }
7  }
```

І отримане повідомлення про помилку від компілятора OpenJDK:

```
F.java:5: error: variable x might be assigned in loop
            x = read();
            ^
1 error
```

Хоча розташування повідомлення є інтуїтивно зрозумілим, це погане пояснення. Проблема полягає не лише в тому, що змінна x призначається в циклі; ця конкретна змінна також позначена як final (рядок 3), а те, що остаточну змінну можна призначити лише один раз. Що, якби замість цього ми отримали наступне повідомлення про помилку:

```
F.java:5: error: The blank final variable "x" cannot
be assigned within the body of a loop that may execute
more than once.
            x = read();
            ^
```

Друге повідомлення дає краще пояснення ніж попереднє. Зокрема, друге повідомлення не лише вказує на наявність проблеми (порожня змінна «x» не може бути призначена»), але й підтверджує це твердження, пропонуючи докази, які пояснюють, в чому це проблема — тому що «x» не можна призначати всередині тіла циклу, який може виконуватися більше одного разу. Тобто друге повідомлення має кращу структуру пояснення, ніж перше. Це повідомлення також надає більш конкретний вміст. На відміну від відносно розпливчастої змінної x у першому повідомленні, у другому повідомленні відразу видно, що x є пустим.

Аргументи — це форма обґрунтування, у якій аргументи використовуються як докази на підтримку висновку. За допомогою теорії аргументації ми можемо оцінити ефективність аргументів. У рамках теорії аргументації модель Тулміна є однією з неформальних моделей міркувань. Зокрема, модель аргументу Тулміна є моделлю макроструктури. Макроструктура досліджує, як компоненти поєднуються для підтримки більшого аргументу; навпаки, мікроструктура перевіряє формулювання та композицію висловлювань «рівня речення». Для ясності ми будемо називати макроструктуру просто структурою, а мікроструктуру – вмістом .

У простій структурі аргументів (рисунок 3.12 а) першим компонентом є твердження, точка зору або судження, яке потрібно обґрунтувати; резолюції також є формою претензії, хоча резолюції необов’язкові в структурі

аргументів. Другий компонент, підстави — це дані, які надають докази цього твердження. Третім компонентом є обґрунтування, яке є сполучною ланкою між підставами та претензією (наприклад, «[претензія] тому що [підстава]»). Разом ці елементи забезпечують простий макет аргументів. Просте розташування аргументів — це мінімальна правильна структура аргументів. Аргументи, які не містять хоча б цих трьох компонентів, вважаються неповноцінними. Специфічними для повідомлень про помилки є вирішення претензій: у першій претензії вказується проблема, а в другій — усунення або виправлення. Хоча це не належні структури аргументів, вони все ж корисні.

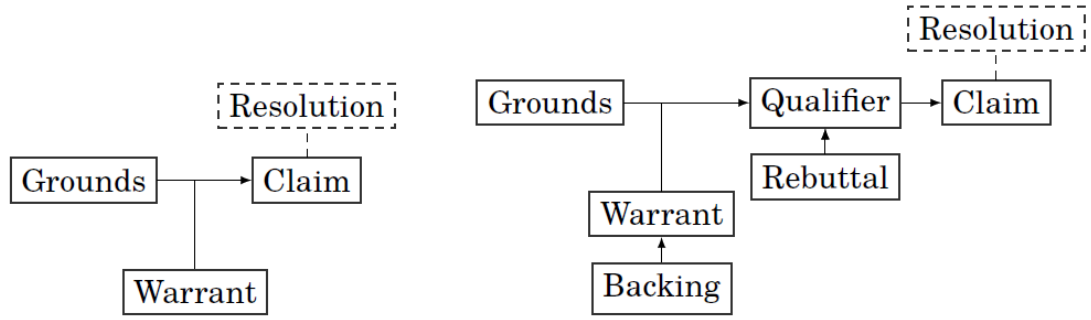


Рисунок 3.12 – Прототип моделі аргументу Тулміна для a) простого розташування аргументів і b) розширеного розташування аргументів

Також існує розширена модель аргументу Тулміна, щоб визначити можливість необхідності вливання додаткових компонентів у простий макет аргументу (рис. 3.12 b). До простих компонентів розкладки аргументів, розширена розкладка аргументів пропонує спростування, коли в аргумент потрібно вставити виняток. Твердження також може бути не абсолютним: у цьому випадку кваліфікуючий компонент може пом’якшити твердження. Нарешті, ордер може бути прийнятий іншою стороною не відразу, і в цьому випадку потрібна додаткова підтримка. Якщо будь-який із цих додаткових компонентів використовується в аргументі, аргумент є розширеною структурою аргументу. Приклад того, як повідомлення про помилку компілятора відображається на структурі аргументів, показано на рисунку

3.13. У цьому прикладі повідомлення про помилку є розширеним аргументом, оскільки воно має підтримку.

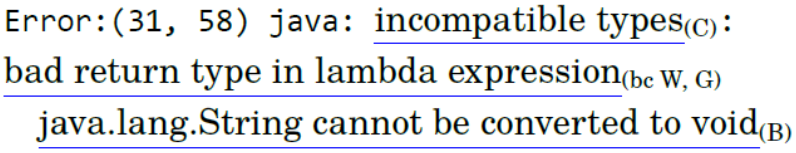


Рисунок 3.13 – Повідомлення про помилку компілятора з Java, анотований компонентами теорії аргументації.

Дане повідомлення містить усі основні компоненти аргументу, щоб задовольнити модель Тулміна: (C) = Претензія, (bc W) = неявне «оскільки» Ордер, (G) = Підстава; розширена конструкція (B).

Розглянемо наступну методику корисності пояснень помилок компілятором.

Питання 1: чи корисні розробникам помилки компілятора, подані як пояснення?

Якщо пояснювальні повідомлення про помилки компілятора корисні для розробників, вони повинні віддавати їм перевагу над повідомленнями про помилки, які є менш пояснювальними у своїй роботі. Якщо це не підтверджено, розробники віддають перевагу представленням повідомлень про помилки на основі інших факторів, таких як велика описовість повідомлення про помилку.

Питання 2: Чим структура пояснень у Stack Overflow відрізняється від повідомлень про помилки компілятора ?

Якщо повідомлення про помилки компілятора та прийняті відповіді Stack Overflow використовують суттєво різні компоненти розташування аргументів, це означає, що структурні відмінності в аргументах відіграють важливу роль розуміння помилки, з якою стикаються розробники.

У той час як деякі підходи до вдосконалення повідомлень про помилки компілятора зосереджуються на вмісті (наприклад, «заплутане

формулювання» в повідомленнях), відмінності в структурі підкреслюють, як компоненти поєднуються для підтримки більшого аргументу, а не самих тверджень.

Крім того, відповідь на це запитання допомагає зрозуміти типи макетів аргументів, які використовуються в прийнятих відповідях. Іншими словами, інструментарі можуть використовувати простір дизайну розташування аргументів для моделювання та структурування повідомлень про помилки автоматичного компілятора для розробників. Простір макета аргументів також можна використовувати як засіб для оцінки наявних повідомлень про помилки та виявлення потенційних прогалин у компонентах аргументів для цих повідомлень.

Питання 3: Чим вміст пояснень у Stack Overflow відрізняється від повідомлень про помилки компілятора ?

Коли макети аргументів структури визначено, вивчення того, як створюються екземпляри компонентів у цих макетах, надає деталі вмісту для інформації, яку розробники вважають корисною в кожному компоненті. Наприклад, одним із способів створення екземпляра підтримки для ордера може бути надання посилання на зовнішню документацію — і якщо ми виявимо, що прийняті відповіді це роблять, майстри інструментів також можуть розглянути можливість включення такої інформації у подання своїх повідомлень про помилки компілятора.

Отримані структури аргументів наведено на рисунках 3.14 і 3.15, відповідно для повідомлень про помилки компілятора та відповідей Stack Overflow.

У цьому квазістатистичному звіті стає зрозуміло, чому схема аргументів для помилок компілятора та прийнятих відповідей Stack Overflow значно відрізняються: повідомлення про помилки компілятора переважно представляють претензію без додаткової інформації та іноді представляють рішення (тобто, виправлення), щоб вирішити проблему.

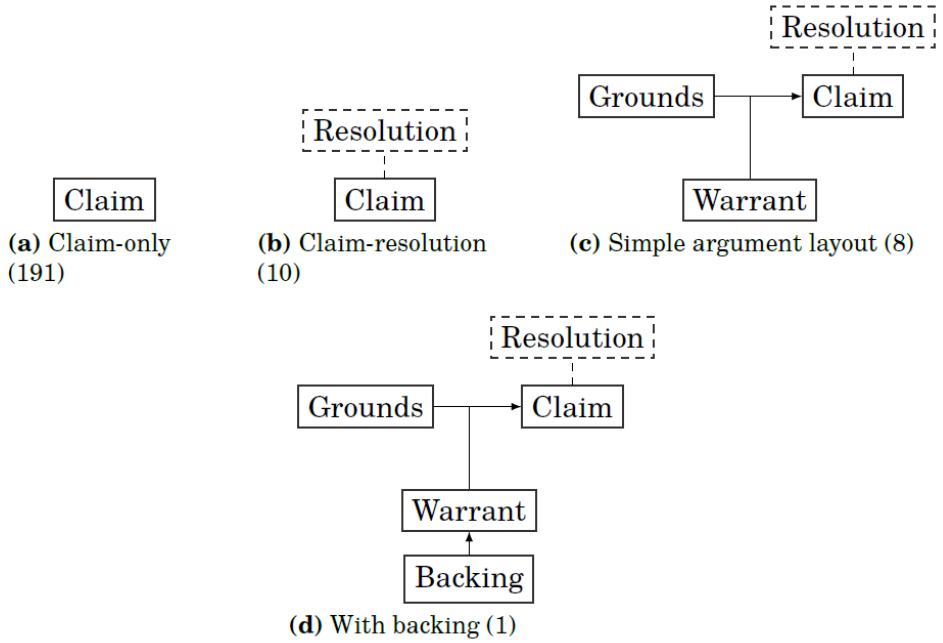


Рисунок 3.14 – Ідентифіковані макети аргументів для повідомлень про помилки компілятора (як знайдено в питаннях Stack Overflow)

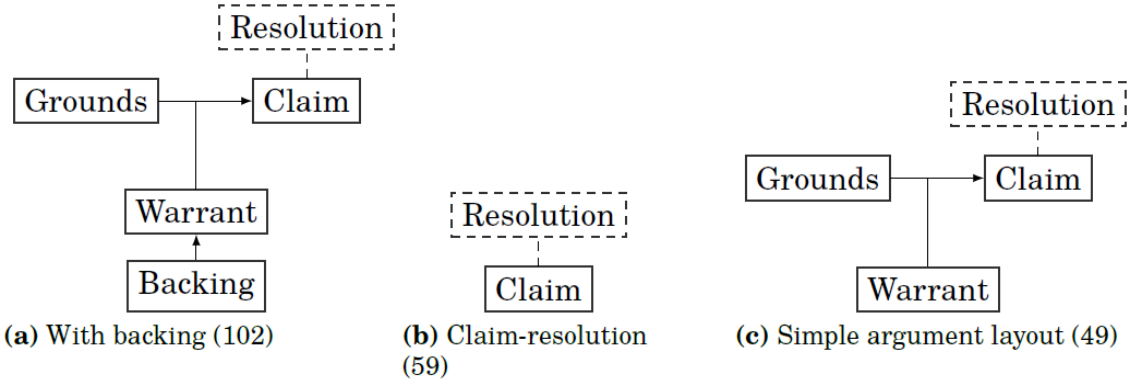


Рисунок 3.15 – Ідентифіковані схеми аргументів для прийнятих відповідей Stack Overflow

На відміну від цього відповіді Stack Overflow дещо інвертуються за частотою розташування аргументів: найпоширеніший макет аргументів розширює простий макет аргументів із підтримкою, за яким йдуть вирішення вимог і простий аргумент із приблизно збалансованою частотою. Під час дослідження не виявлено випадків, коли прийняті відповіді Stack Overflow перефразували б лише повідомлення про помилку компілятора.

Таким чином, прийняті відповіді Stack Overflow не тільки більш узгоджуються з моделлю аргументації Тулміна, ці відповіді задовільно вирішують помилки, коли вихідне повідомлення про помилку компілятора цього не робить.

Висновки до розділу 3

Отже, в цьому розділі представлено дослідження повідомлень про помилки, використовуючи модель аргументації Туліна та форму раціональної реконструкції пояснення. Було досліджено повідомлення про помилки та виконано їх класифікуцію як правильні або неякісні структури аргументів. Результати порівняльної оцінки показали, що розробники віддають перевагу повідомленням про помилку з належною структурою аргументів, а не недоліком аргументів, для вирішення проблеми. Інтуїтивно це має сенс: наявність швидкого рішення, яке б правильно вирішило проблему, може по суті скоротити потребу в раціональній реконструкції.

ВИСНОВКИ

В кваліфікаційній роботі розглянуто процеси підвищення ефективності та оптимальності моделей та методів процесів дебаггінгу, представлено інтерпретації помилок компіляторів.

Виконано огляд інструментів програмного аналізу, який пояснює, чому традиційні таксономії статичного та динамічного аналізу є непридатними для дослідження раціональної реконструкції, і пропонує орієнтовану на презентацію організацію повідомлень про помилки. Описано простір дизайну текстових і візуальних представлень, досліджено літературу щодо емпіричного розподілу повідомлень про помилки інструментів програмного аналізу, що дозволяє визначати пріоритетність удосконалення повідомлень про помилки перед проблемами, з якими насправді стикаються розробники. Наведено хронологію створення рекомендацій щодо дизайну для представлення зручних для людини повідомлень про помилки.

В роботі було досліджено процес використання Eclipse IDE для розуміння та вирішення повідомлень про помилки компілятора Java і виявлено проблеми з інтерпретацією повідомлень про помилки, створених інструментами аналізу програм. Розглянуто концепцію за допомогою якої розробники можуть будували діаграмне представлення повідомлень про помилки, накладене на списки вихідного коду. В результаті було виявлено, що повідомлення про помилки недостатньо узгоджуються з очікуваннями розробників, зокрема щодо виявлення зв’язків між відповідними елементами програми. Було проаналізовано питання та відповіді Stack Overflow, пов’язані з повідомленнями про помилки з використанням теорію аргументації.

Також під час дослідження було виявлено, що повідомлення про помилки, створені людиною, включають додаткові структури розташування аргументів, і що ці структури розташування значно відрізняються від того, як повідомлення про помилки представляють розробникам компілятори.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. D. Abrahams and A. Gurtovoy, C++ Template Metaprogramming: Concepts,Tools, and Techniques from Boost and Beyond. Addison-Wesley, 2004, p. 78.

2. A. Adams, S. Bochner, and L. Bilik, “The effectiveness of warning signs in hazardous work places: Cognitive and social determinants,” Applied Ergonomics, vol. 29, no. 4, pp. 247–254, 1998.

3. A.-R. Adl-Tabatabai and T. Gross, “Source-level debugging of scalar optimized code,” in Programming Language Design and Implementation (PLDI), 1996, pp. 33–43.

4. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd ed. Addison-Wesley, 2007.

5. S. Ainsworth and A. T. Loizou, “The effects of self-explaining when learning with text or diagrams,” Cognitive Science, vol. 27, no. 4, pp. 669–681, 2003.

6. A. Altadmri and N. C. Brown, “37 million compilations: Investigating novice programming mistakes in large-scale student data,” in ACM Technical Symposium on Computing Science Education (SIGCSE), 2015, pp. 522–527.

7. A. Altadmri, M. Kölling, and N. C. C. Brown, “The cost of syntax and how to avoid it: Text versus frame-based editing,” in International Conference on Computers, Software and Applications (COMPSAC), 2016, pp. 748–753.

8. B. de Alwis and G. Murphy, “Using visual momentum to explain disorientation in the Eclipse IDE,” in Visual Languages and Human-Centric Computing (VL/HCC), 2006, pp. 51–54.

9. G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus, “Debugging temporal specifications with concept analysis,” in Programming Language Design and Implementation (PLDI), 2003, pp. 182–195 (see p. 61).

10. C. Angeli, “Diagnostic expert systems: From expert’s knowledge to real-time systems,” in Advanced Knowledge Based Systems: Model, Applications & Research, 2010, ch. 4, pp. 50–73.

11. J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, “PetaBricks: A language and compiler for algorithmic choice,” in Programming Language Design and Implementation (PLDI), 2009, pp. 38 – 49.

12. Apple. Xcode, [Online]. Available: http://developer.apple.com.

13. H. Arksey and L. O’Malley, “Scoping studies: Towards a methodological framework,” International Journal of Social Research Methodology, vol. 8, no. 1, pp. 19–32, 2005.

14. A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel, “TraceBack: First fault diagnosis by reconstruction of distributed control flow,” in Programming Language Design and Implementation (PLDI), 2005, pp. 201–212.

15. N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, “Using static analysis to find bugs,” IEEE Software, vol. 25, no. 5, pp. 22–29, 2008.

16. T. Ball, M. Naik, S. K. Rajamani, T. Ball, M. Naik, and S. K. Rajamani, “From symptom to cause: Localizing errors in counterexample traces,” in Principles of Programming Languages (POPL), vol. 38, 2003, pp. 97–105.

17. T. Ball and S. K. Rajamani, “The SLAM project: Debugging system software via static analysis,” in Principles of Programming Languages (POPL), 2002.

18. T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill, “From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration,” in International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 211–221.

19. T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin, “How should compilers explain problems to developers?” In Foundations of Software Engineering (ESEC/FSE), 2018.

20. T. Barik, K. Lubick, S. Christie, and E. Murphy-Hill, “How developers visualize compiler messages: A foundational approach to notification construction,” in IEEE Working Conference on Software Visualization (VISSOFT), 2014, pp. 87–96.

21. T. Barik, C. Parnin, and E. Murphy-Hill, “One at a time: What do we know about presenting human-friendly output from program analysis tools?” In Evaluation and Usability of Programming Languages and Tools (PLATEAU), 2017.

22. T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, “Do developers read compiler error messages?” In International Conference on Software Engineering (ICSE), 2017, pp. 575–585.

23. T. Barlow and M. S. Wogalter, “Increasing the surface area on small product containers to facilitate communication of label information and warnings,” Proceedings of Interface, vol. 91, no. 7, pp. 88–93, 1991.

24. M. Barr, S. Holden, D. Phillips, and T. Greening, “An exploration of novice programming errors in an object-oriented environment,” in Innovation and Technology in Computer Science Education (ITiCSE), 1999, pp. 42–46.

25. J. A. Bateman and C. Paris, “Phrasing a text in terms the user can understand,” in International Joint Conferences on Artificial Intelligence (IJCAI), 1989, pp. 1511–1517.

26. M. Beaven and R. Stansifer, “Explaining type errors in polymorphic languages,” ACM Letters on Programming Languages and Systems (LOPLAS), vol. 2, no. 1-4.

27. B. A. Becker, G. Glanville, R. Iwashima, C. McDonnell, K. Goslin, and C. Mooney, “Effective compiler error message enhancement for novice programming students,” Computer Science Education, vol. 26, no. 2-3, pp. 148–175, 2016.

28. R. Bednarik and M. Tukiainen, “Temporal eye-tracking data: Evolution of debugging strategies with multiple representations,” in Eye Tracking Research & Applications (ETRA), 2008, pp. 99–102.

29. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, “Explaining counterexamples using causality,” in Computer Aided Verification (CAV), 2009, pp. 94–108.

30. R. F. Beltramini, “Perceived believability of warning label information presented in cigarette advertising,” Journal of Advertising, vol. 17, no. 2, pp. 26 – 32, 1988.

31. D. Benyon and D. Murray, “Applying user modeling to human-computer interaction design,” Artificial Intelligence Review, vol. 7, no. 3, pp. 199–225, 1993.

32. T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, “Efficiency of projectional editing: A controlled experiment,” in Foundations of Software Engineering (FSE), 2016, pp. 763–774.

33. T. Berners-Lee and N. Mendelsohn. (2006). The rule of least power, [Online]. Available: https://www.w3.org/2001/tag/doc/leastPower.html.

34. A. Bessey, D. Engler, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.Henri-Gros, A. Kamsky, and S. McPeak, “A few billion lines of code later: Using static analysis to find bugs in the real world,” Communications of the ACM, vol. 53, no. 2, pp. 66–75, 2010.

35. G. Bierman, M. Abadi, and M. Torgersen, “Understanding TypeScript,” in European Conference on Object-Oriented Programming (ECOOP), 2014, pp. 257–281.

36. D. Binkley, “Source code analysis: A road map,” in Future of Software Engineering (FOSE), 2007, pp. 104–119.

37. M. Birks, Y. Chapman, and K. Francis, “Memoing in qualitative research: Probing data and processes,” Journal of Research in Nursing, vol. 13, no. 1, pp. 68–75, 2008.

38. S. Blackshear and S. K. Lahiri, “Almost-correct specifications: A modular semantic framework for assigning confidence to warnings,” in Programming Language Design and Implementation (PLDI), 2013, pp. 209 – 218.

39. J. M. Bland and D. G. Altman, “Statistical methods for assessing agreement between two methods of clinical measurement,” The Lancet, vol. 327, no. 8476, pp. 307–310, 1986.

40. D. G. Bobrow, S. Mittal, and M. J. Stefik, “Expert systems: Perils and promise,” Communications of the ACM, vol. 29, no. 9, pp. 880–894, 1986.

41. M. D. Bond, G. Z. Baker, and S. Z. Guyer, “Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses,” in Programming Language Design and Implementation (PLDI), 2010, pp. 13–24 (see p. 64).

42. P. N. van den Bosch, “A bibliography on syntax error handling in context free languages,” ACM SIGPLAN Notices, vol. 27, no. 4, pp. 77–86, 1992.

43. B. D. Boulay and I. Matthew, “Fatal error in pass zero: How not to confuse novices,” Behaviour & Information Technology, vol. 3, no. 2, pp. 109–118, 1984.

44. N. Boustani and J. Hage, “Improving type error messages for generic Java,” Higher-Order and Symbolic Computation, vol. 24, no. 1-2, pp. 3–39, 2011.

45. C. C. Braun, N. C. Silver, and B. R. Stock, “Likelihood of reading warnings: The effect of fonts and font sizes,” Proceedings of the Human Factors and Ergonomics Society Annual Meeting, vol. 36, no. 13, pp. 926–930, 1992.

46. C. Bravo-Lillo, L. F. Cranor, J. Downs, and S. Komanduri, “Bridging the gap in computer security warnings: A mental model approach,” IEEE Security & Privacy, vol. 9, no. 2, pp. 18–26, 2011.

47. G. Brooks, G. J. Hansen, and S. Simmons, “A new approach to debugging optimized code,” in Programming Language Design and Implementation (PLDI), 1992, pp. 1–11.

48. R. Brooks, “Towards a theory of the cognitive processes in computer programming,” International Journal of Human-Computer Studies, vol. 51, no. 2, pp. 197–211, 1977.

49. “Towards a theory of the comprehension of computer programs,” International Journal of Man-Machine Studies, vol. 18, no. 6, pp. 543–554, 1983.

50. P. J. Brown, “Error messages: The neglected area of the man/machine interface,” Communications of the ACM, vol. 26, no. 4, pp. 246–249, 1983.

51. S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, “It’s alive! Continuous feedback in UI programming,” in Programming Language Design and Implementation (PLDI), 2013, pp. 95–104.

метадані

Заголовок

Підвищення ефективності та оптимальності моделей та методів процесів дебагінгу

Автор

Борщ В.В. Науковий керівник / Експерт

підрозділ

King Danylo University

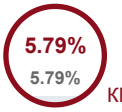
Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

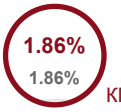
Заміна букв	ß	4
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	43

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



КП 1



КП 2

25
Довжина фрази для коефіцієнта подібності 2

14991
Кількість слів

113078
Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	http://gustavopinto.org/lost+found/scam2019.pdf	38	0.25 %
2	https://dl.acm.org/doi/10.1145/3212695	36	0.24 %
3	Enhancing Python Compiler Error Messages via Stack Overflow Christoph Treude,Emillie Thiselton;	35	0.23 %
4	The role of semiotics in health, safety, and environment communication in South African mining and its influence on organizational culture M.A. Goncalves,L. Scott;	35	0.23 %