МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 46.00.00.000 ПЗ

Група ІПм-23-2

Лешко Владислав

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Лешко Владислав Вікторович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі, методи та алгоритми покращення ефективності шаблонів

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Лешко В.В.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Шекета Василь Іванович, д.т.н., професор

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

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

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**

доц. В.В. Бандура

“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Лешку Владиславу Вікторовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Моделі, методи та алгоритми покращення ефективності шаблонів”

керівник проекту (роботи) Шекета Василь Іванович, д.т.н., професор

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних технологій обробки шаблонів проектування ПЗ

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

1. Дослідження предметної області використання моделей та шаблонів проектування ПЗ
2. Дослідження шаблонів як рішення для генерації коду з моделі. Варіативність
3. Дослідження залежностей в об’єктній мові
4. Моделі, методи та підходи покращення ефективності використання шаблонів

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

1. Модель введення для прикладу шаблонів (рис. 2.1)
2. Шаблон стратегії (рис. 2.2)
3. Приклад діаграми функцій калькулятора (рис. 2.3)
4. Позначення діаграми мінливості (рис. 2.4)
5. Приклад діаграми мінливості калькулятора (рис. 2.5)

6. Консультанти розділів проекту (роботи)

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

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Дослідження предметної області використання моделей та шаблонів проектування ПЗ	15.10.2024	виконано
4	Дослідження шаблонів як рішення для генерації коду з моделі. Варіативність	08.11.2024	виконано
5	Дослідження залежностей в об’єктній мові	20.11.2024	виконано
6	Моделі, методи та підходи покращення ефективності використання шаблонів	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 80 с., 16 рис., 50 джерел.

Тема: Моделі, методи та алгоритми покращення ефективності шаблонів

Об’єкт дослідження: процес автоматизованої генерації програмного коду на основі шаблонів.

Мета роботи: дослідити та обґрунтувати методи та алгоритми, які підвищують ефективність використання шаблонів у генерації коду шляхом інтеграції синтаксичних правил і стандартів кодування.

Предмет дослідження: моделі, методи та алгоритми підвищення ефективності використання шаблонів у генерації програмного коду.

Результати дослідження

В роботі розроблено алгоритми для обробки мови варіантів, які забезпечують відповідність згенерованого коду стандартам кодування, а також запропоновано стратегії для ефективного використання шаблонів у процесі генерації коду.

Висновок

Запропоновані методи дозволяють уникнути клонування коду та розсіяного коду, покращуючи зручність супроводу та повторного використання шаблонів.

ГЕНЕРАЦІЯ КОДУ, ШАБЛОНИ, ПРОМІЖНА МОВА ВАРІАНТІВ, ВАРІАТИВНІСТЬ, СИНТАКСИЧНІ ПРАВИЛА, МОДЕЛЬНО-ОРІЄНТОВАНА РОЗРОБКА, АЛГОРИТМИ ОБРОБКИ

ABSTRACT

Master Thesis: 80 pp., 16 fig., 50 sources.

Thesis Subject: Models, methods and algorithms for increasing the efficiency of templates

The object of research: the process of automated generation of software code based on templates.

The goal of the work: to investigate and justify methods and algorithms that increase the efficiency of using templates in code generation by integrating syntactic rules and coding standards.

The subject of research: models, methods and algorithms for increasing the efficiency of the use of templates in the generation of software code.

Research results

Algorithms for processing language variants are developed in the work, which ensure compliance of the generated code with coding standards, and strategies are also proposed for the effective use of templates in the code generation process.

Conclusion

The proposed methods allow you to avoid code cloning and distributed code, improving the ease of maintenance and reuse of templates.

CODE GENERATION, TEMPLATES, INTERMEDIATE LANGUAGE OF OPTIONS, VARIABILITY, SYNTAX RULES, MODEL-ORIENTED DEVELOPMENT, PROCESSING ALGORITHMS

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ART - Adaptive Reuse Technique

AST - Abstract Syntax Tree

BFS - Breadth First Search

DSL - Domain Specic Language

EMF - Eclipse Modeling Framework

IVL - Intermediate Variant Language

MDE - Model Driven Engineering

MTL - Model to Text Language

OCL - Object Constraint language

OMG - Object Management Group

SPL - Software Product Line

XVCL - XML-based Variant Conguration Language

ВСТУП

Актуальність теми.

Сучасна індустрія розробки програмного забезпечення стикається з необхідністю створення все більш складних, гнучких і адаптивних систем, що потребує ефективних інструментів для автоматизації процесів генерації коду. Шаблони як інструмент розробки відіграють ключову роль у стандартизації та повторному використанні кодових рішень, однак їх ефективність часто обмежується проблемами, пов'язаними з варіативністю, такими як клонування умов, надлишковий об’єктний код і розсіяні залежності. Ці проблеми не лише ускладнюють підтримку та масштабування програмного забезпечення, а й призводять до збільшення витрат на розробку та супровід.

Особливого значення набуває інтеграція синтаксичних правил і стандартів кодування безпосередньо в шаблони, що дозволяє зменшити кількість помилок і забезпечити якість програмного продукту. На цьому фоні актуальним є впровадження проміжних мов для моделювання варіативності, розробка нових алгоритмів генерації коду та стратегій управління шаблонами, які здатні подолати існуючі обмеження та підвищити ефективність розробки.

Крім того, актуальність теми зростає в умовах поширення підходів модельно-орієнтованої розробки (Model-Driven Engineering, MDE), де шаблони є основним механізмом перетворення моделей у програмний код. Успішне вирішення проблем варіативності та оптимізація роботи з шаблонами сприяють підвищенню продуктивності процесів розробки, знижують витрати на створення та адаптацію програмних продуктів, а також розширюють можливості для інновацій у галузі програмної інженерії.

Таким чином, дослідження спрямоване на вдосконалення методів роботи з шаблонами є актуальним як для академічної спільноти, що

займається розробкою нових технологій, так і для практиків, які прагнуть підвищити ефективність своїх програмних рішень.

Мета дослідження - дослідити та обґрунтувати методи та алгоритми, які підвищують ефективність використання шаблонів у генерації коду шляхом інтеграції синтаксичних правил і стандартів кодування.

Об'єкт дослідження - процес автоматизованої генерації програмного коду на основі шаблонів.

Предмет дослідження - моделі, методи та алгоритми підвищення ефективності використання шаблонів у генерації програмного коду.

Задачі дослідження:

- Проаналізувати проблеми варіативності в шаблонах, зокрема явну та неявну розсіяну умову, клоновані умови та надлишковий об’єктний код.

- Розробити підхід до моделювання структури та відношень між фрагментами коду за допомогою проміжної мови варіантів.

- Запропонувати алгоритми для обробки проміжної мови варіантів та генерації коду, що відповідає стандартам кодування.

- Обґрунтувати стратегії обробки шаблонів для підвищення їхньої ефективності.

Методи дослідження

В роботі використано теоретичний аналіз існуючих методів та інструментів роботи з шаблонами та мовами програмування; vоделювання структури та відношень між фрагментами коду; алгоритмічний підхід до розробки та реалізації методів обробки шаблонів; емпіричний аналіз ефективності розроблених алгоритмів на тестових прикладах.

Наукова новизна отриманих результатів

Запропоновано підхід до інтеграції синтаксичних правил та стандартів кодування в шаблони на основі проміжної мови варіантів, яка дозволяє усунути проблему клонування умов і розсіяного коду.

Практичне значення результатів

Розроблені методи та алгоритми можуть бути інтегровані в існуючі інструменти автоматизованої генерації коду, що дозволить зменшити витрати на розробку програмного забезпечення, підвищити його якість і полегшити процес супроводу. Запропонований підхід може бути використаний в освітніх програмах для навчання розробників принципам ефективної роботи з шаблонами.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 80 сторінок, і містить 20 рисунків, 5 таблиць, список використаних джерел із 50 найменувань.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ВИКОРИСТАННЯ МОДЕЛЕЙ ТА ШАБЛОНІВ ПРОЕКТУВАННЯ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Особливості використання модельно-орієнтованої розробки в IT індустрії

Модельно-орієнтована розробка (MDE) – це підхід до розробки програмного забезпечення, який ставить в основу моделі. Замість того, щоб писати код вручну, розробники створюють моделі, які представляють систему, що розробляється, на високому рівні абстракції. Ці моделі потім використовуються для генерації коду, документації та інших артефактів.

Часто в IT компаніях є ряд програмних проектів де використовують інструменти модельно-орієнтованої розробки (MDE). Ці інструменти використовують моделі та перетворення моделей для генерації програмного забезпечення. Існує два різних види перетворень моделей: перетворення "Модель в Модель" та перетворення "Модель в Текст".

Моделями користуються щодня, іноді не знаючи, що моделями користуються. Наприклад, технічні креслення будинку є макетами будинку. Вони не показують весь будинок, тому що технічний малюнок, наприклад, не показує, якого кольору стіни.

Модель є абстрактною, оскільки включає не всю інформацію, яка існує в реальності. Наприклад, технічні креслення того, як побудувати будинок, є абстракцією справжнього будинку. Фактичні будівельні блоки будинку, наприклад цегла, не включені в креслення. Можна створити декілька різних моделей для однієї системи, наприклад, технічне креслення будинку включає креслення, яке визначає розміри кімнат, а також креслення, яке визначає електричні кабелі. Моделі не обов’язково мають бути візуальними, наприклад, той самий будинок також можна описати як список матеріалів [2].

Моделі можна використовувати для опису того, що повинна робити система або як вона створюється. Моделі можна порівнювати з попередніми версіями, а потім використовувати для відображення змін. Інше важливе використання моделей полягає в тому, що моделі можуть бути зрозумілі людям з різними дисциплінами. Кожен може переглянути модель, щоб перевірити, чи відповідає вона очікуванням цієї особи. Завдяки різноманітним способам використання моделей у спілкуванні вони набули популярності. Більшість програмних проектів використовують моделі щодня з причин, перерахованих вище.

Околиці чи міста також можна описати моделями. Така модель не містить такого рівня деталізації, як технічні креслення будинку. Модель мікрорайону – це вищий рівень абстракції, ніж технічні креслення будинку. У розробці програмного забезпечення V-модель [3] може бути використана як процес розробки. На рисунку 1.1 подано представлення V-моделі.

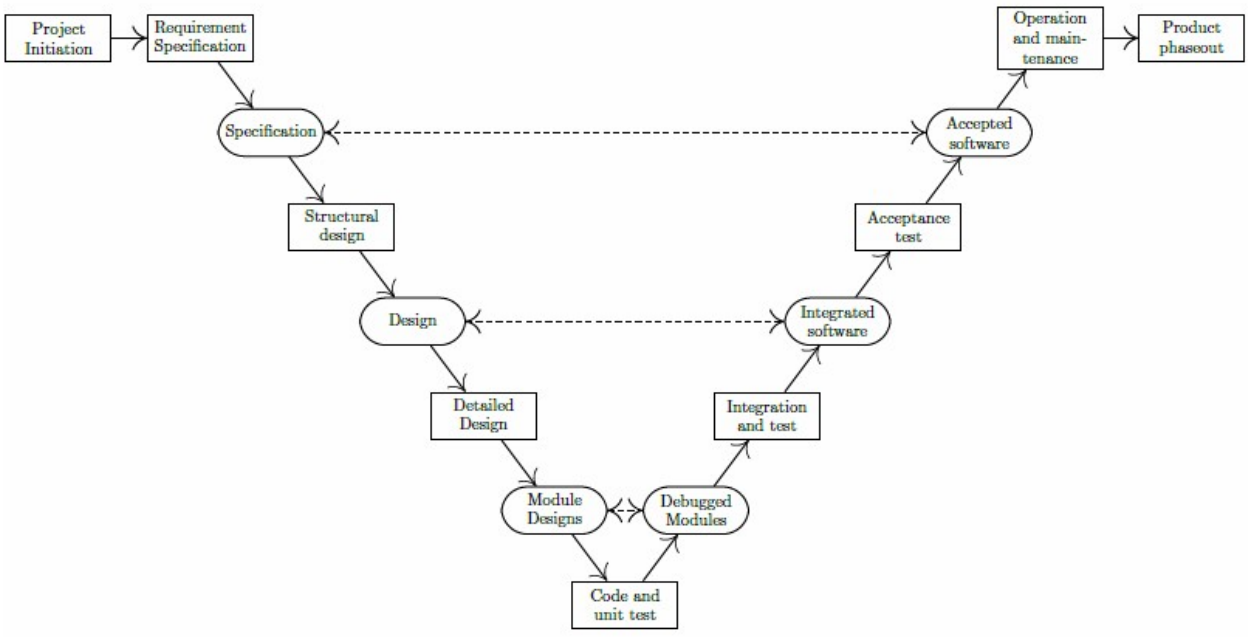

Рис. 1.1. Представлення V-моделі для розробки

На кожному кроці вниз V-моделі створюється одна або більше моделей, які описують ту саму систему на різних рівнях абстракції. Кроки вниз у V-моделі є трансформаціями моделі, де рівень абстракції знижується з

кожним кроком, поки не буде досягнуто впровадження продукту. Реалізація в розробці програмного забезпечення зазвичай базується на коді. Трансформації також можуть підвищити рівень абстракції шляхом видалення інформації з моделі. Перетворення моделі також можуть трансформувати моделі, зберігаючи той самий рівень абстракції.

Для будинку створюється кілька моделей, щоб визначити кожну деталь цього будинку. Модель абстрагує інформацію від системи, але кілька моделей можуть зберігати всю необхідну інформацію. Будуючи будинок, він будується з використанням інформації з моделей. Цей процес однаковий для кожного будинку. Процес будівництва починається з фундаменту. Після чого зводяться стіни. Процес будівництва закінчується дахом (щоб побудувати будинок, потрібно зробити більше, але це спрощений процес). Цей процес виконується для кожного будинку з тією лише різницею, що вхідні моделі змінюються. Цей процес можна назвати трансформацією макета в реальний будинок. Автоматизуючи більшість перетворень, можна уникнути багатьох повторюваних і схильних до помилок роботи.

Використання моделей і перетворення моделей для автоматизації розробки програмного забезпечення називається моделюванням, керованим інженерією (MDE). Трансформація в MDE існує у двох формах: модель у модель і модель у текст. Існують різні мови для перетворення моделі в модель, такі як Henshin [3], ATL [20], ETL [22] або QVTo [26]. Модель у текст також називають генератором коду. Деякі генератори коду створюють частини проекту або навіть можуть створити весь проект.

У цій роботі основна увага приділяється перетворенням "Модель в Текст", зокрема перетворенням, які реалізуються за допомогою шаблонів. Шаблони - це фрагменти, які ще не завершені. Механізм шаблонів може приймати модель та шаблон як вхідні дані та генерувати документ або код у випадку MDE. У шаблоні існують дві різні мови: мова об'єкта та метамова. Мова об'єкта - це мова, яка генерується з шаблону. Метамова - це мова, яка використовується для визначення шаблону та його дій.

У цій роботі визначено шість різних проблем з шаблонами. У перших двох проблемах відношення в об'єкті, мова створює клоновані умови. Третя проблема випливає з перших двох, де ймовірність того, що програміст зробить помилку, зростає через відношення в мові об'єкта та клоновані умови. Правила в програмному проекті існують для підвищення читабельності та зручності супроводу кодових баз. Ці правила є правилами угоди про об'єктний код. Прикладом такого правила є те, що коментарі перед функцією повинні пояснювати, що робить функція. Четверта проблема полягає в тому, що правила угоди не відокремлені від шаблонів. Тому правила угоди потрібно реалізовувати кілька разів. П'ята проблема виникає в шаблонах, коли алгоритм має кілька варіантів у можливих кількох вимірах. Це зменшує огляд алгоритму та його варіантів для розробника. Шоста і остання проблема полягає в тому, що об'єктний код повинен бути узгодженим у своїх елементах, наприклад, прототип функції в C++ повинен бути таким самим, як і реалізація.

Перші дві проблеми аналізуються шляхом створення одностороннього відображення на діаграму мінливості. На цій діаграмі показано, що відношення відсутнє в метамові, тоді як воно присутнє в мові об'єкта. Явне створення цього відношення в шаблоні вирішило б проблему. Це робиться за допомогою проміжної мови між метамовою та мовою об'єкта, яка називається Intermediate Variant Language (IVL). IVL має два компоненти: варіанти та вимоги. З його допомогою можна явно вказати відношення між елементами. Якщо це відношення вказано, то воно може автоматично підтримуватися, що має покращити підтримку шаблону. Оскільки IVL є проміжною мовою, її можна обробляти окремо після метамови. Для обробки IVL представлено та обговорено два різних алгоритми. Оскільки третя проблема випливає з перших двох, вона автоматично вирішується за допомогою цього рішення.

Четверта проблема полягає в тому, що правила угоди реалізуються кілька разів. Це можна вирішити, відокремивши різні аспекти від шаблону. У

цих шаблонах існують три різні аспекти: вміст, що надається шаблоном, вміст, що надається моделлю, та вміст, що надається правилами угоди. Ці три різні аспекти розділені та змодельовані. Створюючи шаблон, який спеціально реалізує правила угоди, можна використовувати ту саму кодову базу з кількома правилами угоди або реалізувати те саме правило угоди для кількох кодових баз. Це підвищує модульність і має покращити зручність супроводу.

1.2. Постановка проблеми дослідження

Проекти програмного забезпечення та мови мають певний спосіб структурування своїх елементів, наприклад файлів, операторів або команд. Наприклад, включення файлу C++ завжди знаходяться у верхній частині файлу, а список використовуваних бібліотек вказується в Makefile. Ці правила походять з архітектурних правил організації або проекту. Стандарти кодування визначають багато правил [14, 17, 35]. Іншим джерелом цих правил є сама мова. У Python, наприклад, оператор імпорту можливий лише на початку файлу.

Ми розглядаємо компанію яка має декілька проектів програмного забезпечення, результатом яких є інструменти MDE для різних клієнтів. Результат інструментів MDE має відповідати стандартам кодування.

Ці правила необхідні для покращення читабельності та організації коду. Вони застосовуються до кожного файлу проекту, і часто ці правила застосовуються до кількох проектів. Наприклад, в організації, яка має кілька проектів, можуть застосовуватися однакові архітектурні правила.

У випадку написаного вручну коду всі елементи у файлі мають відношення один до одного, зокрема відношення перебування в одному файлі. У разі генерації коду вміст одного отриманого файлу коду може походити з кількох джерел. Це може призвести до інакше змодельованого коду, що зазвичай неможливо. Наприклад, за допомогою підшаблонів можна додати декілька рядків на початок кожного створеного файлу вихідного коду,

наприклад ліцензії на файл. Це може призвести до кращого моделювання коду, але оскільки існує кілька джерел, це також збільшує складність. На додаток до того факту, що кілька джерел вносять свій внесок, деякі частини результату можуть бути необов'язковими. Для цього може знадобитися наявність інших частин. Це може не тільки збільшити швидкість створеної програми та зменшити час компіляції, але також може переконатися, що деякі бібліотеки не використовуються. Це може бути корисним, коли бібліотеки з якихось причин недоступні або непотрібні.

У процесі генерації файлу вихідного коду можливо, що інформація про те, що має бути згенеровано, стане доступною лише пізніше. Тож можливо, що не все доступне під час створення певної частини файлу. Крім того, не кожен фрагмент коду включений у вихідні дані генератора, є можливість для додаткових частин. Відсутність інформації або факт наявності необов'язкових частин у вихідних даних може призвести до недотримання мовних або архітектурних правил. Вирішення цих проблем безпосередньо в генераторі коду може призвести до проблем з обслуговуванням, оскільки, коли потрібна інформація, можуть бути введені клони коду або частина коду розкидана між різними джерелами.

Одним із способів генерації коду є використання шаблонів. Неформальне визначення шаблонів — це документ із пропусками, які потрібно заповнити даними або результатами дії [29]. Ці проблеми створюються за допомогою мови шаблонів.

Маленькі та прості шаблони не мають проблем, тому що огляд шаблону легко отримати. Коли розмір шаблону збільшується, стає складніше отримати огляд шаблону. У великих проектах кодова база з шаблонами може стати досить великою. Проект може мати базу коду до 30 тисяч рядків коду, що містить до 200 файлів із 200 підгенераторами коду. Крім того, стек викликів із 5–10 підшаблонів не є чимось незвичайним у цих проектах.

У постановці проблеми вводяться правила та їх проблеми в процесі генерації. Головне питання дослідження : чи можливо включити синтаксичні

правила та стандарти кодування в текстові шаблони, не створюючи клонів коду або розрізненого коду.

Щоб відповісти на це питання дослідження, визначено три підзапитання дослідження.

1. Яку структуру та які відносини важко виразити шаблонною мовою.

Ці структури та відносини мають бути певним чином змодельовані таким чином, щоб це могло потенційно покращити технічне обслуговування генератора.

2. Чи можна цю структуру та ці відносини змоделювати мовою шаблонів.

Якщо це певним чином змодельовано, цю модель слід обробити, щоб залишити лише оригінальну цільову мову.

3. Як можна розв’язати модель, щоб вона розв’язувалася оригінальною цільовою мовою.

1.3. Генератори коду. Використання шаблонів для генерації коду

Генератори коду — це програми, які виводять код на основі певних вхідних даних, наприклад моделі. Ці генератори використовуються в процесі розроблення на основі моделі (MDE) для створення коду. Генератор коду використовує для виведення певну мову. Після запуску з певним введенням він генерує речення з цієї мови. Мова виводу називається об'єктною мовою. Мова, на якій написаний генератор коду, називається метамовою [5]. Якщо ці дві мови однакові, то це називається генератором однорідного коду. Якщо мови різні, це називається гетерогенним генератором [5].

Генератори однорідного коду мають перевагу в тому, що не додають складності іншої мови, але за визначенням обмежені в об’єктній мові. Ще одна велика перевага полягає в тому, що перевірка синтаксису та перевірка типу є безкоштовною, оскільки шаблони є частиною самої об’єктної мови. Перевірка синтаксису та перевірка типу призводить до того, що помилки

виявляються раніше в конвеєрі. Прикладами таких мов є MetaML [39] і шаблони в C++ [37]. У лістингу 1.1 показано приклад шаблонів C++. Шаблон визначається в рядках з 6 до 9. Коли ці шаблони використовуються в поєднанні з певним типом, генерується код T. Отже, у рядках 14, 18 і 22 код є створений для цього типу. Для типів int, double і string відповідно.

Лістинг 2.1: Приклад шаблонів C++

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  template <typename T>
7  inline T const& Max (T const& a, T const& b)  {
8    return a < b ? b:a;
9  }
10
11 int main () {
12   int i = 39;
13   int j = 20;
14   cout << "Max(i, j): " << Max(i, j) << endl;
15
16   double f1 = 13.5;
17   double f2 = 20.7;
18   cout << "Max(f1, f2): " << Max(f1, f2) << endl;
19
20   string s1 = "Hello";
21   string s2 = "World";
22   cout << "Max(s1, s2): " << Max(s1, s2) << endl;
23
24   return 0;
25 }
```

Генератори різнорідного коду мають інший об’єкт і метамову. Додана мова збільшує складність, але є більше свободи в тому, яку мову вона може створити. Через свободу об’єктної мови перевірка синтаксису не є безкоштовною [5]. Існує кілька різних концепцій генерації гетерогенного коду.

Абстрактне синтаксичне дерево. Генератор коду може використовувати абстрактне синтаксичне дерево (AST) [30]. AST реалізовано за допомогою строго типізованого інтерфейсу прикладного програмування (API), який описує об’єктний код. Метакод може створити AST у будь-якій формі. За допомогою цього API можна створити розбірник, який генерує рядок із довільного AST.

Такий тип генераторів може досягати високої складності. У порівнянні з іншими видами понять, це дуже багатослівний спосіб генерації коду. Згенерований код синтаксично правильний, оскільки API строго типізований. Дерево завжди правильно створене. В AST немає ні інформації про розмітку коду, ні про коментарі. Отже, такі типи елементів не можуть бути згенеровані таким генератором коду.

Зрештою, код — це просто великий рядок, який також можна створити за допомогою операторів друку (або щось на зразок конструктора рядків). Ці генератори коду можуть генерувати будь-яку об’єктну мову. Оскільки метамова є мовою загального призначення, а не доменно-специфічною мовою (DSL), генератор складніше повторно використовувати або адаптувати. Прикладом генератора на основі операторів друку є ApiGen [7].

Переписування термінів базується на розпізнаванні термінів і подальшій їх заміні іншим терміном [6]. Переписування термінів розглядається як технологія з крутою кривою навчання [21]. Розуміння об’єктного коду також складне, оскільки за своєю суттю весь об’єктний код розкиданий навколо термінів. Це також зроблено в [38].

Останньою формою генерації коду є текстові шаблони. Шаблони відомі в різних областях. Одним із простих прикладів шаблонів є шаблон електронної пошти, де, наприклад, ім’я можна заповнити пізніше. Це використовується, наприклад, зі списками розсилки, де потрібно надіслати багато листів, які відрізняються лише невеликою частиною. Розширений тип шаблонів — це шаблони, які використовуються у веб-розробці для створення екземплярів HTML для створення динамічних веб-сторінок. У контексті MDE шаблони використовуються для створення екземпляра коду. Приклад такого шаблону показано в лістингу 1.2.

Механізми шаблонів були створені з метою спростити генерацію коду. Це може бути HTML у випадку веб-розробки або мови загального призначення під час створення програми. Ці механізми мають DSL як

метамову, яка використовується для реалізації генератора. Механізм аналізує шаблон і виконує шаблон із знанням домену.

Лістинг 1.2. Приклад MTL

```
[module generateIf('http://example.org/model')]
[template public generateElementIf(aData : Data)]
[file (aData.Name, false, 'UTF-8')]

[if(aData.UseFeature1)]
from .ComponentA import ComA
[/if]

def funcA():
  print('Hello')
  [if(aData.UseFeature1)]
  print(ComA.value())
  [/if]

[/file]
[/template]
```

Шаблони – це документи з пропусками, які потрібно заповнити даними або результатами дії [29]. Через використання DSL багато файлу, який буде згенеровано, уже видно у вихідному шаблоні. Двигун має замінити лише заповнювачі. Одним із стимулів для розробки шаблонів є те, що перегляд має бути відокремлений від моделі [29].

Як і інші генератори коду, існує мова об’єктів і метамова. У шаблонах коду ці дві мови переплітаються.

Більшість метамов і механізмів мають загальний набір функцій. У наведеному нижче списку показано деякі з цих функцій [5]:

• Виведення в один файл

• Доступ до моделі введення

• Підшаблони

• Збіг-заміна

Більшість метамов також мають наступні функції, які можна замінити конструкціями підшаблонів і заміни відповідності [5]. Ці функції інтуїтивно зрозумілі в мета-мові, тому більшість їх має.

• Підстановка зі змінної

• Якщо структура

• Для кожного циклу

Існують відмінності між різними метамовами та механізмами. Наступні функції є функціями, які реалізовані не всіма мовами (цей список не є вичерпним):

• Кілька вихідних файлів [27]

• Неможливі розрахунки в рамках шаблону

• Відсутність побічних ефектів при виконанні шаблону [29]

Неможливість обчислень у межах шаблону не здається функцією. Це правда, оскільки це робить мову менш потужною, але проблема з обчисленнями полягає в тому, що до шаблону також може бути додана бізнес-логіка. У роботі Парра визначено індекс заплутаності [29]. Індекс заплутаності показує, наскільки зображення переплутано з моделлю. Нижчий індекс означає меншу взаємозв’язок між моделлю та видом. П'ять критеріїв визначено в наступному списку [29]:

1. Представлення не може модифікувати модель безпосередньо змінюючи об’єкти даних моделі або викликаючи методи моделі, які викликають побічні ефекти.

2. Представлення не може виконувати обчислення на основі залежних значень даних.

3. Перегляд не може порівнювати залежні значення даних.

4. Подання не може робити припущення типу даних.

5. Дані з моделі не повинні містити інформацію про відображення чи макет.

Індекс розраховується шляхом підрахунку кількості порушених правил. правило 2 визначає, що жодні обчислення не повинні бути можливими в рамках метамови. Автор стверджує, що всі обчислення мають відбуватися в моделі. Правило номер п'ять завжди порушується, тому що інформація з моделі необхідна для візуалізації перегляду [29]. Отже, мінімальний індекс заплутаності дорівнює одиниці.

Одне, що слід зазначити, це те, що в більшості механізмів шаблонів немає знань про мову об’єктів. Він агностик будь-якої об'єктної мови, а

об'єктна мова просто розглядається як текст. Значною перевагою цього є те, що механізм шаблонів легко адаптується для кожної об’єктної мови.

1.3.1. Мета мови

За роки веб-розробки було розроблено кілька різних метамов. Більшість метамов розроблено для веб-розробки, але інші, такі як Model to Text Language [27] і Xpand [44], розроблені спеціально для MDE. Було створено лише невелику кількість мов, де індекс заплутаності не перевищує одиниці. Прикладами мов, які мають індекс заплутаності одиниці, є Repleo [5], StringTemplates [36] і Mustache [25]. Наступна модель для мови тексту обговорюється далі.

Рис. 1.2. Ресурс StringTemplates

Мова моделювання тексту (MTL) [27] використовується в цій роботі як метамова. Цю метамову можна використовувати в середовищі MDE Eclipse Modeling Framework (EMF). MTL реалізовано Acceleo [1], який є одним із підтримуваних інструментів генерації коду в EMF.

Acceleo - це інструмент з відкритим кодом для генерації коду, розроблений Eclipse Foundation. Він дозволяє використовувати модельно-орієнтований підхід до створення застосунків. Acceleo реалізує стандарт "MOFM2T" від Object Management Group (OMG) для виконання перетворення "Модель в Текст".

Основні можливості:

- Acceleo дозволяє генерувати будь-який тип вихідного коду з будь-якого джерела даних, доступного у форматі EMF (Eclipse Modeling Framework).

- Використовує шаблони для визначення того, як код має бути згенерований з моделі. Шаблони можуть бути налаштовані для задоволення конкретних потреб.

- Acceleo інтегрований з Eclipse IDE, що забезпечує зручне середовище для розробки генераторів коду.

- Інструмент може генерувати код на різних мовах програмування, включаючи Java, C++, Python та інші.

- Acceleo є проектом з відкритим кодом, що дозволяє його вільно використовувати та модифікувати.

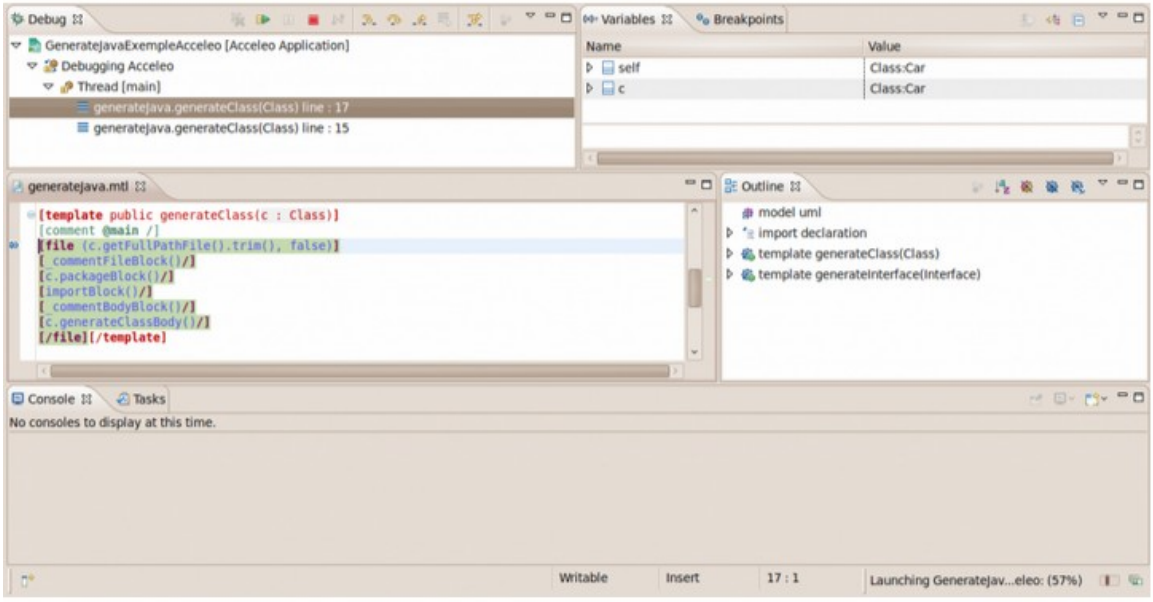

Рис. 1.3. Налагоджувач Acceleo

Мова розроблена та підтримується групою керування об’єктами (OMG). MTL використовує мову обмежень об’єктів (OCL) [28] як мову вираження значень із моделі, а також умов. У цьому розділі пояснюються лише елементи мови, які використовуються для цієї роботи. Для повного пояснення мови MTL існує специфікація, яка підтримується OMG [27].

Висновки до розділу

У першому розділі розглянуто особливості використання модельно-орієнтованої розробки (MDD) у сучасній IT-індустрії, визначено основні переваги цього підходу, такі як підвищення ефективності розробки, автоматизація процесів і забезпечення високого рівня абстракції. Проаналізовано постановку проблеми дослідження, яка полягає в необхідності оптимізації процесів розробки програмного забезпечення шляхом впровадження моделей і шаблонів, що сприяють стандартизації та повторному використанню рішень. Окрему увагу приділено генераторам коду, які дозволяють автоматизувати створення програмного коду на основі шаблонів і забезпечують скорочення витрат на ручну працю, зменшуючи кількість помилок. Розглянуто мету використання мов опису моделей і шаблонів, які виступають ключовими інструментами для ефективної реалізації концепції модельно-орієнтованої розробки. Цей підхід визначено як перспективний і здатний істотно змінити традиційні підходи до розробки програмного забезпечення.

РОЗДІЛ 2. ДОСЛІДЖЕННЯ ШАБЛОНІВ ЯК РІШЕННЯ ДЛЯ ГЕНЕРАЦІЇ КОДУ З МОДЕЛІ. ВАРІАТИВНІСТЬ.

2.1. Проблеми які виникають при використанні шаблонів

Коли MDE використовується в промисловості, розроблений процес стає більш складним. Ця складність проявляється в більшій кількості перетворень моделі та більш складних перетвореннях моделі. У шаблонах коду ця складність проявляється в більшій кількості шаблонів і більших шаблонах. Як і будь-який проект програмного забезпечення, більші шаблони можуть призвести до проблем, які ускладнять обслуговування. У цьому розділі розглядаються різні проблеми, які існують у шаблонах. Для кожної задачі наведено один або декілька прикладів. Приклади написані мовою MTL [27] як метамовою. Об'єктною мовою в прикладах є Python або C/C++. Модель введення прикладів шаблонів у цьому розділі описана на рисунку 2.1. Ця модель є чисто вигаданою, де об'єкт Data використовується як основний об'єкт. Об'єкт Data містить нуль або більше об'єктів DataB.

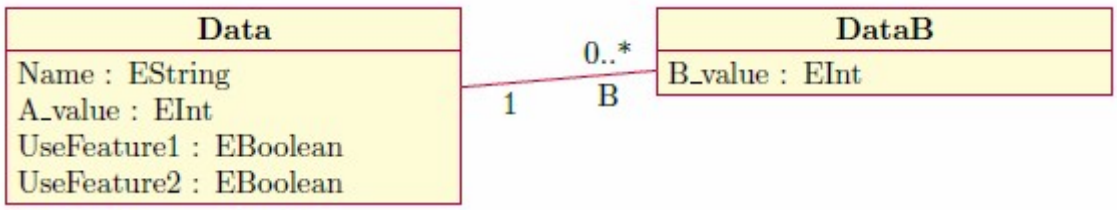

Рис 2.1. Модель введення для прикладу шаблонів

Конвенційні правила часто ігноруються під час генерації вихідного коду, оскільки технічне обслуговування виконується в шаблоні та моделі, тому отриманий код можна легко відновити. Тому більшість описаних проблем виглядають на перший погляд дуже штучно. Згенерований код із прикладів у цьому розділі навіть не призводить до помилок, що посилює аргумент.

Можна стверджувати, що конвенційні правила насправді корисні у згенерованому коді. Якщо, наприклад, згенерований код використовується або вручну адаптується розробниками, тоді добре мати правильно відформатований код із правильними коментарями. Також у деяких випадках, коли згенерований код потрібно перевірити в репозиторій з кодом, розробленим вручну. Політика компанії може стверджувати, що весь код, який надходить у репозиторій, має відповідати стандартам. Отже, згенерований код також має відповідати стандартам.

У прикладах у цьому розділі проблема неочевидна, тому що загальний огляд прикладів легко отримати. Причина цього здебільшого полягає в тому, що прикладів досить мало. У галузевому проекті база коду більша. Проект може мати базу коду до 30 тисяч рядків коду, що містить до 200 файлів із 200 шаблонами. Крім того, стек викликів із 5 - 10 підшаблонів не є рідкістю в цих проектах.

2.1.1. Явна розсіяна умова

Перша проблема PS 1 існує в шаблоні коду, де частина об’єктного коду залежить від іншої частини об’єктного коду. Наприклад, у лістингу 2.1, де рядок 12 використовує ComA.vaiue(). ComA визначається з .ComponentA імпортувати ComA онлайн 6. Фрагменти об'єктного коду мають відношення def-use. Проблема виникає, якщо вихідний об’єктний код необов’язково включений у вихідні дані. Стан вихідного коду (у прикладі на рядку 11) має відповідати цільовому коду (рядок 5 у прикладі).

Лістинг 2.1. PS1 – явна розсіяна умова

```
1  [module generateIf('http://example.org/model')]
2  [template public generateElementIf(aData : Data)]
3  [file (aData.Name, false, 'UTF-8')]
4
5  [if(aData.UseFeature1)]
6  from .ComponentA import ComA
7  [/if]
8
9  def funcA():
10   print('Hello')
11   [if(aData.UseFeature1)]
12   print(ComA.value())
13   [/if]
14
15 [/file]
16 [/template]
```

У невеликому файлі (наприклад, лістинг 2.1), де гарний огляд легко створити, це не є безпосередньою проблемою. У більш загальному випадку, як показано в лістингу 2.2 проблема ускладнюється. У цьому прикладі знову використовується MTL як мета-мова, а C++ і пакетний сценарій використовуються як об’єктні мови. Повторно створено два різні файли. Перший — це файл коду C++, а другий — пакетний сценарій, який компілює перший згенерований файл.

Лістинг 2.2. PS1 – Явна розсіяна умова з кількома файлами

```
1  [module generateIf('http://example.org/model')]
2
3  [template public generateElementIf(aData : Data)]
4  [file (aData.Name, false, 'UTF-8')]
5  #include <iostream>
6  [if(aData.UseFeature1)]
7  #include <ComA.h>
8  [/if]
9
10 void Main() {
11   std::cout << "hello";
12   [if(aData.UseFeature1)]
13   std::cout << ComA.value();
14   [/if]
15   std::cout << std::endl;
16 }
17 [/file]
18 [/template]
19
20 [template public generateMakefile(aData : Data)]
21 [file ('build.bat', false, 'UTF-8')]
22 g++ \
23 [if(aData.UseFeature1)]
24 -IComA \
25 [/if]
26 [aData.Name /] -o [aData.Name /].exe
27 [/file]
28 [/template]
```

Тепер об'єктний код в рядку 13 лістингу 2.2 залежить не від одного рядка, а від двох рядків, які генеруються у двох різних файлах, зокрема включення файлу заголовка (рядок 7) і аргумент компонувальника на компіляторі (рядок 2).

Отже, виходячи з наведеного вище лістингу 2.2 можна стверджувати, що оператор if у метакоді копіюється тричі. Коли проект розростається, ці два шаблони, ймовірно, будуть розділені на окремі модулі. Якщо розбити, то складніше підтримувати ці умови, оскільки немає явного зв’язку між частинами вихідного коду.

2.1.2. Неявна розсіяна умова

Задача PS2 є більш складним варіантом PS1. Умова в PS1 (aData.UseFeature1) явно показана в прикладах і скопійована в різні місця. У промислових проектах це не завжди так. Використання об'єктного коду, який має залежності, може бути оточене кількома вкладеними циклами for і операторами if. Це можна поєднати з підшаблонами, які можуть підтримувати певну форму поліморфізму [2]. Кілька вкладених циклів for, операторів if і підшаблонів призводять до більш складних і складних умов. Це визначає, чи частина об'єктного коду, яка має залежності, включена до згенерованого виводу. Тільки ця умова явно не вказана в шаблоні.

Лістинг 2.. PS 2 – умова неявного розсіяння

```
1  [module generateImplicitIf('http://example.org/model')]
2
3  [template public generateElementImplicitIf(aData : Data)]
4    [file (aData.Name, false, 'UTF-8')]
5      [if(aData.UseFeature1 && aData.B->notEmpty() && aData.B.any(b | b.B_value > 0))]
6      from .ComponentA import ComA
7      [/if]
8      def funcA():
9        print("Hello")
10       [if(aData.UseFeature1)]
11       print("world")
12       [showB(aData.B)/]
13       [/if]
14   [/file]
15 [/template]
16
17 [template public showB(singleInstanceB : DataB)]
18   [if(singleInstanceB.B_value > 0)]
19     print(ComA.Calculate([singleInstanceB.B_value /]))
20   [/if]
21 [/template]
```

Приклад цієї проблеми показано в лістингу 2.3. У цьому прикладі об'єктний код, який має залежність, знаходиться в рядку 19, де використовується ComA. Клас ComA визначено в рядку 6, а рядок 19 безпосередньо оточений оператором if. Цей оператор if знаходиться всередині імені підшаблону showB, який включено в рядок 12, що оточений оператором if. На перший погляд, використання ComA залежить лише від двох умов. У семантиці MTL зазначено, що якщо функція має один параметр, але викликається зі списком, то визначається неявний цикл. У цьому прикладі aData.B — це список DataB, тоді як тип параметра singleInstanceB —

single DataB. Під час виклику шаблону онлайн існує неявний цикл for в рядку 12 .

Отже, у прикладі використання ComA оточене двома операторами if і циклом for. Умова в якій в рядку 19 додається набагато складніше, ніж у попередніх прикладах PSl. Ще можна сформулювати умову: aData.UseFeature1 && aData.b->notEmpty()&& aData.b.any(b | b. mvalue > 0). Якщо додається більше підшаблонів і більше вкладених операторів if і циклів for, то умову стає дедалі складніше написати та зрозуміти. Навіть якщо умова написана правильно, при зміні шаблону виникає проблема. Розробнику важко визначити наслідки зміни стосунків залежностей між різними частинами об’єктного коду.

Результатом труднощів у написанні та розумінні умови є те, що під час клонування умови будуть зроблені помилки. Це призведе до того, що умови, які повинні бути однаковими, виявляються сильнішими, слабшими або непорівнянними одна з одною. Якщо з якоїсь причини бібліотека недоступна (наприклад, пропрієтарна або не підтримується на платформі) і умова, на якому рядку 6 має бути включено сильніше, ніж умова, на якому рядку 19 включено, може призвести до помилок. І навпаки, якщо умова включення слабша, ніж її використання, це призведе до помилок, оскільки елемент об’єктного коду не визначено.

2.1.3. Проблема надлишкового об'єктного коду

Проблема, яку створює PS2 полягає в тому, що рядки можна додавати двічі, оскільки в умовах зроблено помилки. Якщо умови прості, це не буде проблемою, але якщо умови стають складнішими, як у PS 2, розробник програмного забезпечення хоче розділити ці умови. Це може створити більш зручний шаблон, але можуть виникнути помилки, і це може створити можливість включити той самий рядок двічі. Приклад цієї проблеми показано в лістингу 2.4. В рядках 7 і 10 виконується той самий імпорт, але

цей імпорт включається у вихід залежно від різних умов, aData.UseFeaturel і aData.UseFeature2 відповідно.

Лістинг 2.4. PS4 – надлишковий код

```
1  [module generateRedundant('http://example.org/model')]
2
3  [template public generateElementRedundant(aData : Data)]
4  [file (aData.name, false, 'UTF-8')]
5
6  [if(aData.UseFeature1)]
7  from .ComponentA import ComA
8  [/if]
9  [if(aData.UseFeature2)]
10 from .ComponentA import ComA
11 [/if]
12
13 def funcA():
14   print("Hello")
15   [if(aData.UseFeature1)]
16   print(ComA.value())
17   [/if]
18   [if(aData.UseFeature2)]
19   print(ComA.value2())
20   [/if]
21
22 [/file]
23 [/template]
```

Штучний характер цього прикладу також виникає через той факт, що виконання того самого імпорту двічі (рядок 7 і 10 у лістингу 2.4) часто вирішується під час виконання або під час компіляції. Це вірно для операторів import (Python) або include (C/C++) , але два однакові визначення функції призводять до помилок.

2.1.4. Конвенційні правила

У середовищах програмування існують правила для підвищення читабельності та зручності обслуговування кодових баз. Походження цих правил здебільшого базується на кодових угодах. Ці кодові угоди написані для коду, розробленого вручну, а не для згенерованого коду. Наприклад, у шаблонах коду важко підтримувати відступи, оскільки вкладені оператори if і цикли метамови також будуть мати відступи. Ще один вид правил — це правила коментарів, які мають бути написані перед елементами мови об’єктів. Виділені рядки в лістингу 2.5 та 2.6 – це рядки коментарів, які розповідають про програмний елемент(и).

Лістинг 2.5. PS4 - Приклад умов кодування A

```
"""
Description: -Generated-
Version: -Generated-
Authors: -Generated-
History: -Generated-
Date: 23-05-
Copyright
"""

#---------------------------
# imports
#---------------------------
from .ComponentA import ComA

#---------------------------
# function definition
#---------------------------
def [aData.Name /](x):
  """Execute [aData.name /]
  In:
  x -- x position

  Out:
  z -- some description
  """
  print("Hello")
  return ComA.calculate(x)

#---------------------------
# execution
#---------------------------
print([aData.Name /]([aData.A_value /]))
```

Лістинг 2.6. PS4 - Приклад умов кодування B

```
"""
Description: -Generated-
Version: -Generated-
Authors: -Generated-
History: -Generated-
Date: 23-05
Copyright :
"""

#---------------------------
# imports
#---------------------------
from .ComponentB import ComB

#---------------------------
# function definition
#---------------------------
[for(b : DataB | aData.B)]
def Exec[b.B_value /]():
  """Execute Exec[b.B_value /]
  In:
  --

  Out:
  z -- some description
  """
  return ComB.calculate([b.B_value /])
[/for]

#---------------------------
# execution
#---------------------------
print([aData.Name /]([aData.A_value /]))
```

Наприклад, перші вісім рядків списку описують інформацію про файл. У лістингу 2.5 від рядка 19 до рядка 25 описано інформацію про функцію. Ці коментарі повторюються в обох файлах. Виділені рядки в лістингу 2.5 і 2.6 в основному однакові, за винятком деяких відхилень.

Повторне використання таких шаблонів складно. Якщо в іншому проекті можна використовувати той самий шаблон, шаблони потрібно змінити, оскільки він не відповідає тим самим правилам коду, що й оригінальний проект, тому потрібно створити ручну копію та підтримувати її з іншим набором стандартів кодування. З іншого боку, структурну частину шаблону також важко повторно використовувати, оскільки вона переплітається з фактичним вмістом.

2.1.5. Різні варіації алгоритму

Для деяких алгоритмів існують різні варіанти алгоритму. Наприклад, можна реалізувати кілька варіантів алгоритму сортування, які реалізують різні типи, наприклад рядки та цілі числа, дивіться лістинг 2.7, 2.8 і 2.9 .

Лістинг 2.7. PS5 - Загальний алгоритм

```
1  [template public quicksort(V: int)]
2  void quicksort([type(V) /] *A, int len) {
3    if (len < 2) return;
4    [type(V) /] pivot = A[len / 2];
5    int i, j;
6    for (i = 0, j = len - 1; ; i++, j--) {
7      while ([compareFirst(V) /]) i++;
8      while ([compareSecond(V) /]) j--;
9      if (i >= j) break;
10     [type(V) /] temp = A[i];
11     A[i]      = A[j];
12     A[j]      = temp;
13   }
14   quicksort(A, i);
15   quicksort(A + i, len - i);
16 }
17 [/template]
18 [template public type(V: int)]
19   [if(V == 0)][int_type() /][/if]
20   [if(V == 1)][char_type() /][/if]
21 [/template]
22 [template public compareFirst(V: int)]
23   [if(V == 0)][int_compareFirst() /][/if]
24   [if(V == 1)][char_compareFirst() /][/if]
25 [/template]
26 [template public compareSecond(V: int)]
27   [if(V == 0)][int_compareSecond() /][/if]
28   [if(V == 1)][char_compareSecond()/][/if]
29 [/template]
```

У випадку цього прикладу існує лише один вимір варіації, але це також можливо в кількох вимірах, що експоненціально збільшує кількість можливих реалізацій. Ці варіанти часто розбиваються на різні модулі та файли. Під час редагування алгоритму або варіантів важко отримати огляд того, що відбувається, оскільки кожен файл має бути актуальним разом з двома іншими. У разі більшої кількості варіантів кількість файлів збільшується на стільки ж, а складність також зростає.

Лістинг 2.8. PS5 – Варіант 1

```
[template public int_type()]
  int
[/template]

[template public int_compareFirst()]
  A[i] < pivot
[/template]

[template public int_compareSecond()]
  A[j] > pivot
[/template]
```

Лістинг 2.9: PS 5 – Варіант 2

```
[template public char_type()]
  char*
[/template]

[template public char_compareFirst()]
  strcmp(A[i], pivot) < 0
[/template]

[template public char_compareSecond()]
  strcmp(A[j], pivot) > 0
[/template]
```

Існує паралель із шаблоном проектування стратегії з шаблоном проектування: елементи багаторазового об’єктно-орієнтованого програмного забезпечення [13]. Схема стратегії показана на рисунку 2.2. Там за допомогою поліморфізму можна створити кілька різних варіантів одного алгоритму. Це робиться за допомогою класів стратегії. Стратегічні класи мають ряд методів. Ці методи є паралельними підшаблонам прикладу шаблону в списках 2.8 і 2.9.

Ця проблема виходить за межі цього дослідження, тому в даній дипломній роботі ця проблема не вирішується. Щоб вирішити цю проблему,

необхідно створити кращий вигляд для розробника алгоритму з його варіантами. Можливо, це можна зробити в редакторі коду або в іншій програмі.

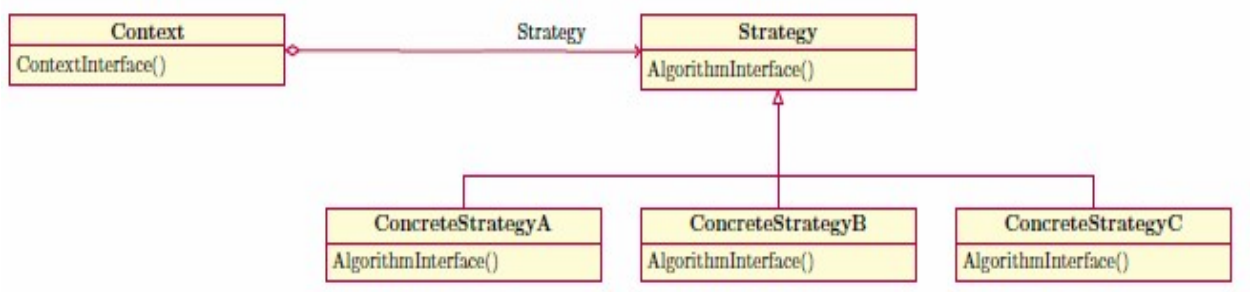

Рис. 2.2. Шаблон стратегії [13]

2.1.6. Проблема розсіяного об'єктного коду

Проблема PS6 також має кілька умов, клонованих як PS1, але між різними доповненнями є узгодженість. Успадкування та синтаксичні правила можуть вимагати, щоб визначення однієї функції, а іноді навіть оголошення, визначали кілька разів. Наприклад, у лістингу 2.10 інтерфейс визначено в лістингу 2.11 клас визначено в лістингу 2.12 визначено тіло функції. Усі ці фрагменти коду мають (майже) однакове визначення методу, але різну семантику. Ці три фрагменти також мають однакову умову, aData.UseFeaturei. Бажано, щоб ці визначення були поруч, але розкидані в трьох різних файлах.

Лістинг 2.10. PS 6 – визначення інтерфейсу

```
1  class IDraw {
2    [if(aData.UseFeature1)]
3    virtual void DrawCircle(int x, int y, int size) = 0;
4    [/if]
5  }
```

Лістинг 2.11. PS 6 – визначення класу

```
1  class BMPDraw: IDraw {
2    [if(aData.UseFeature1)]
3    void DrawCircle(int x, int y, int size);
4    [/if]
5  }
```

Лістинг 2.12. PS 6 – визначення тіла

```
[if(aData.UseFeature1)]
void BMPDraw::DrawCircle(int x, int y, int size){
  [comment implementation /]
}
[/if]
```

Щоб вирішити цю проблему, об’єктну мову слід додатково проаналізувати в поєднанні з метамовою.

Отже, шаблони представлені в цьому розділі є рішенням для генерації коду з моделі. Представлено мову шаблону MTL, що використовується для прикладів.

2.2. Дослідження залежностей в об’єктній мові. Концепція розробки Software Product Line

До промислової революції кожен продукт створювався вручну та підлаштовувався під замовлення. У революції було введено поняття товарної лінії. Ця зміна була ініційована, щоб задовольнити більший попит споживачів на продукцію. Сьогодні багато споживачів хочуть, щоб їхній продукт був налаштований певним чином, наприклад, змінити зовнішній вигляд або функціональність. Для досягнення цієї мети в лінійці продуктів були введені варіації. Це відкрило шлях до масової кастомізації [32].

Software Product Line (SPL), або лінійка програмних продуктів, - це підхід до розробки програмного забезпечення, який фокусується на створенні сімейства пов'язаних програмних систем з використанням спільного набору активів та керованого набору варіацій

Продуктова лінія щоразу виробляє той самий продукт. У розробці програмного забезпечення це вже можливо шляхом дублювання програмного забезпечення. Введення варіацій у лінійку продуктів створює спосіб персоналізувати продукт. Це також можна ввести в розробку програмного забезпечення, щоб продукт був налаштований для клієнтів.

Щоб спеціалізувати програмне забезпечення для клієнта, варіативність повинна бути введена під час розробки програмного забезпечення. Це

робиться в SPL [32]. SPL складається з двох частин: процесу розробки домену та процесу розробки програми. У процесі розробки домену визначаються загальні частини програми. У процесі розробки програми використовуються результати процесу розробки домену, а до результатів додаються налаштування.

У процесі проектування домену вказуються загальні риси та варіації систем. У цьому процесі також створюються нормальні артефакти розвитку. Відмінність від розробки окремого продукту полягає в тому, що вибір продукту може бути зроблений, що є мінливістю продукту. Тому необхідні додаткові анотації до артефактів, щоб уточнити, що частини належать до різновиду продукту. Якщо анотація не вказана, належить до загальної частини товару.

Точки, де можна зробити вибір, є точками варіації. Будь-яка точка варіації має один або кілька варіантів, які можуть бути необов'язковими або обов'язковими. Точки варіацій і варіанти можуть існувати в будь-якому зазвичай використовуваному артефакті програмного забезпечення, такому як код, документація та моделі.

Фактична програма, яку отримує клієнт, створюється процесом розробки програми. Цей процес використовує артефакти процесу розробки домену. У цих артефактах вказані варіанти, для яких потрібно зробити вибір. Набір варіантів для точок варіації для певного продукту називається конфігурацією [32].

2.2.1. Моделювання варіативності

Існує кілька методів, щоб показати мінливість у моделі, у цій роботі показано два способи: діаграма ознак і діаграми мінливості [32].

Діаграми функцій показують дерево функцій, які можна включити або виключити з програмного забезпечення. Приклад діаграми функцій програми-калькулятора зображено на рисунку 2.3. Діаграма функцій може описати всі функції програми, а також те, яка функція необов'язкова, а яка

потрібна. Наприклад, калькулятор має обов'язкову функцію «Обчислення та інтерфейс», але функція «Показати графік» необов'язкова. У функції «Обчислення» можна вибрати різні типи обчислень, наприклад, прості, розширені, функції або диференціальні рівняння. Це також те, який тип інтерфейсу має мати програма. У графі відображення є більше додаткових функцій.

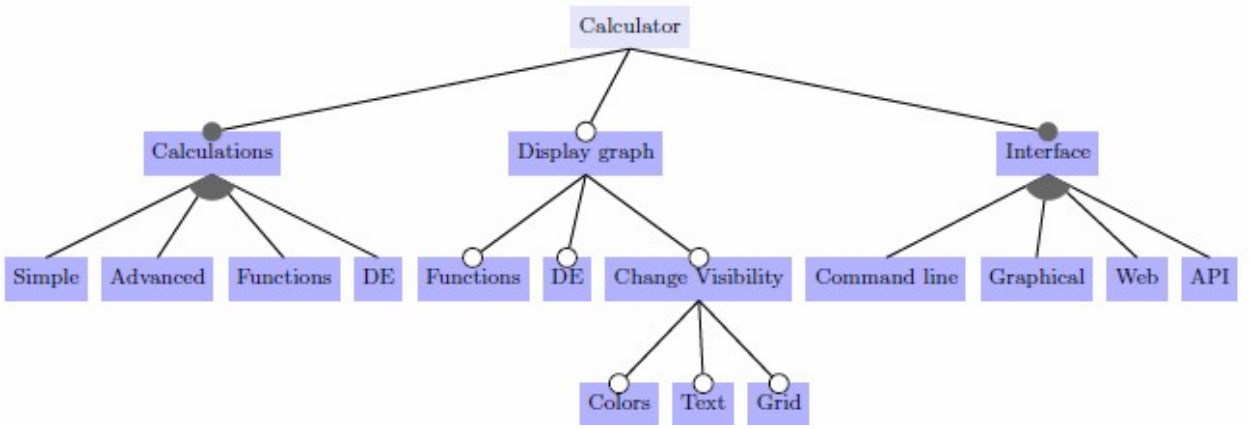

Рис. 2.3. Приклад діаграми функцій калькулятора

Недоліком діаграми функцій є те, що вона може показати різницю між альтернативними функціями, які є спільними для всіх програм, і альтернативними функціями, які можна вибрати окремо для конкретної програми. Моделювання мінливості в моделі ознак може призвести до неправильної інтерпретації [32]. Більше того, на діаграмі ознак відсутній механізм групування, який би дозволяв довільні ознаки призначати деяким варіантам [32].

Іншим позначенням для опису мінливості є діаграма мінливості [32]. Позначення діаграми мінливості наведено на рисунку 2.4, а приклад діаграми показано на рисунку 2.5. Діаграми вказують точки варіації з певною назвою. Точка варіації може мати від одного до багатьох варіантів. Варіант також пов'язаний принаймні з однією точкою варіації. Точка варіації та варіант можуть мати необов'язковий або обов'язковий зв'язок. Якщо варіант

необов'язковий, то є можливість вказати альтернативний вибір з мінімальною та максимальною потужністю.

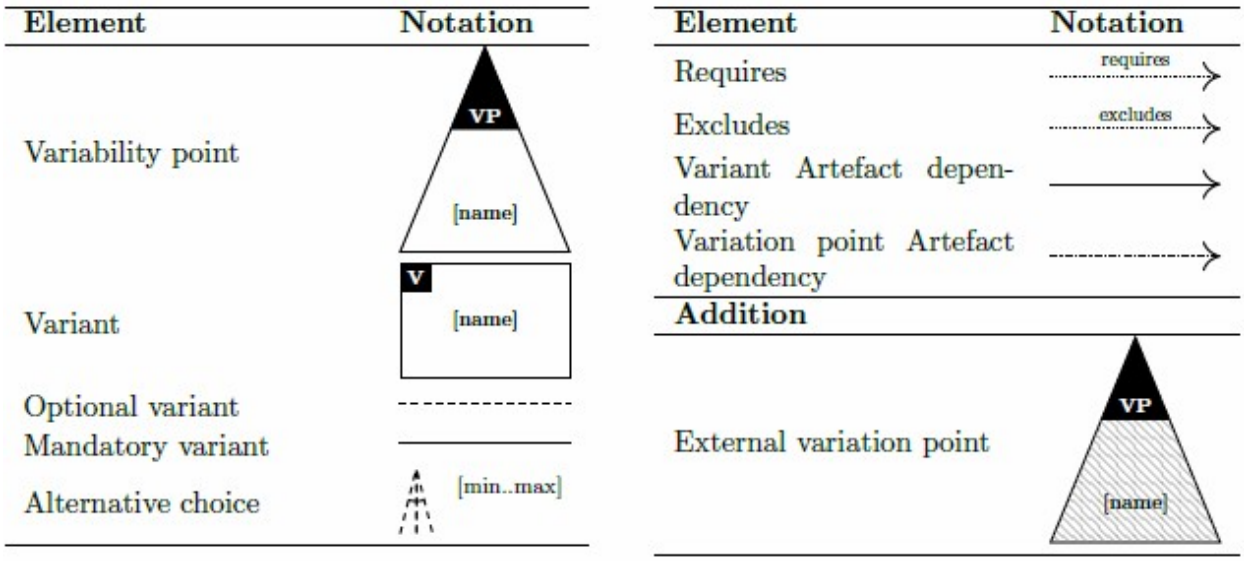

Рис. 2.4. Позначення діаграми мінливості

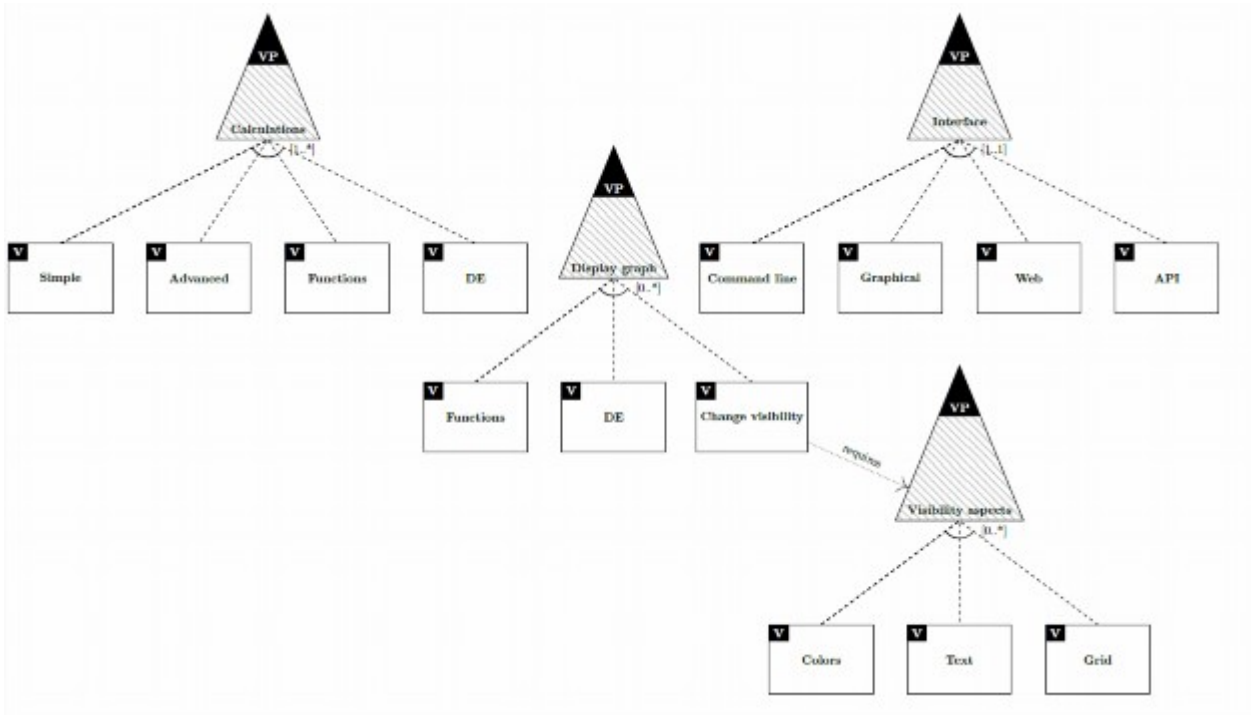

Рис. 2.5. Приклад діаграми мінливості калькулятора

Точки варіації та варіанти можуть мати залежності від інших точок варіації та варіантів, ці залежності моделюються за допомогою вимог. Також

відношення виключення можна змоделювати на діаграмах змінності, що гарантує, що елементи ніколи не вибираються разом. Останнім відношенням, яке можна моделювати на діаграмах змінності, є залежність артефакту. Ці діаграми є ортогональними в розробці проекту, що означає, що діаграми можна використовувати на кожному етапі процесу розробки паралельно з артефактами розробки, які зазвичай використовуються. За допомогою залежностей артефактів певні частини артефактів можна включити або виключити залежно від того, обраний варіант чи ні. Приклад змінності із залежностями артефакту показано на рисунку 3.1.

Одним доповненням до діаграм мінливості, зазначених у [32], є представлення внутрішніх і зовнішніх точок варіації. Зовнішня мінливість визначається тим, який вибір є видимим для клієнта.

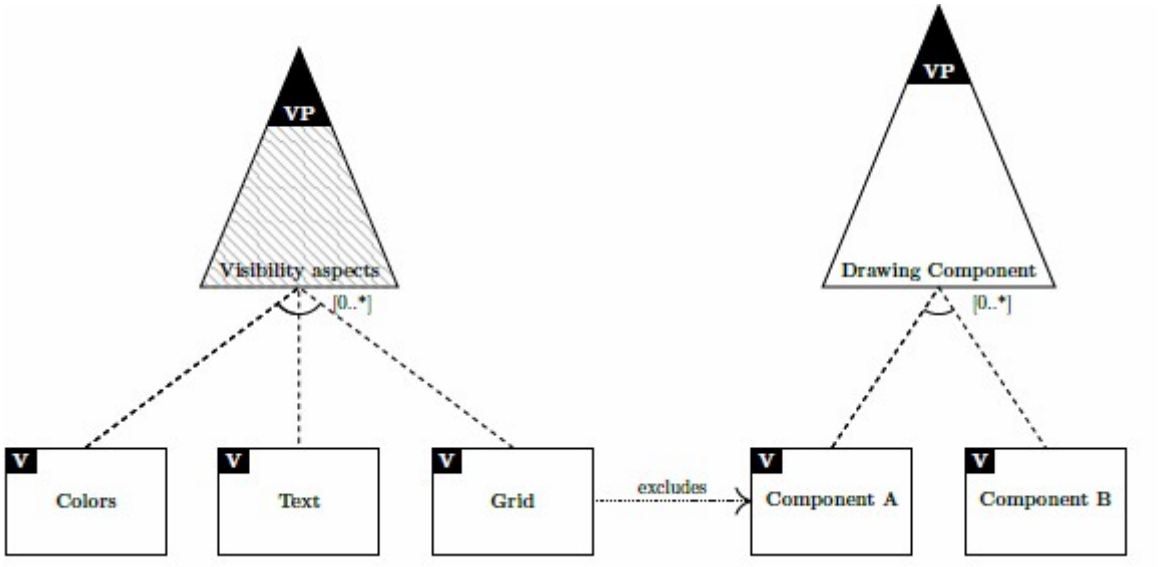

Рис. 2.6. Приклад діаграми мінливості з внутрішніми виборами

Клієнта цікавить лише вибрана кількість варіантів. Нецікаві частини програмного забезпечення все ще необхідні для забезпечення роботи програмного забезпечення. Наприклад, в калькуляторі необов'язково змінювати видимість графіка. Це цікаво замовнику, тому це зовнішня мінливість. Прикладом внутрішньої мінливості є те, який компонент використовуватиметься для малювання на графіку. Приклад цього показано

на рисунку 2.6. Клієнтів цікавить лише те, що намальовано на екрані, але для вибору деяких варіантів необхідно також зробити вибір у компоненті, який малює графік. Якщо точка мінливості невидима або не цікава клієнту, то вона описує внутрішню мінливість.

2.2.2. Мінливість у шаблонах

Якщо шаблон коду описує мінливість згенерованого коду, то можна відобразити діаграму мінливості шаблону коду. Ряд елементів метамови є точками мінливості в шаблоні, але є кілька цікавих елементів: if, if-else, цикл for-each та підстановка змінної.

Семантика оператора if диктує, що частина коду включається або виключається на основі умови. Передбачається, що ця умова отримує інформацію лише з моделі, тому це зовнішня точка мінливості. Оскільки вона може включати або виключати деяку частину результату, її можна змоделювати за допомогою точки мінливості та одного необов'язкового варіанта, який належить до цієї точки мінливості.

Оператор if-else - це вибір між двома варіантами. Це знову базується на умові, яка використовує інформацію з моделі. Отже, це знову зовнішня точка мінливості. Оскільки в цьому виборі є два варіанти, вводяться два варіанти. Можливий лише один, а не обидва. Це можна змоделювати за допомогою альтернативного вибору. Оператор if-else - це точка мінливості з двома варіантами з альтернативним вибором, де потрібно вибрати рівно один.

Підстановку змінної складніше змоделювати на такій діаграмі, оскільки може бути підставлений будь-який рядок, який може бути в моделі. Якщо припустити, що немає умов щодо значення в моделі, можна підставити нескінченну кількість значень. Тоді це призведе до нескінченної кількості варіантів. Отже, це призведе до необмеженої моделі. Це відхилення від визначення мінливості, де точки мінливості завжди мають скінченну кількість варіантів. Це відхилення необхідне, оскільки мова шаблонів є потужнішою, ніж звичайне визначення мінливості.

Цикл for-each - це повторення деякого коду. Кількість повторень залежить від моделі. Отже, for-each також є зовнішньою точкою мінливості.

Оскільки не можна робити припущень щодо обмежень моделі, слід припустити, що список, по якому ітерується цикл for-each, не має обмежень. Отже, кількість ітерацій також не обмежена, отже, кількість варіантів необмежена. Варіанти, що належать до цієї точки мінливості, представляють ітерації. Отже, цикл for-each представляє точку мінливості з нескінченною кількістю варіантів. Це знову відхилення від визначення мінливості.

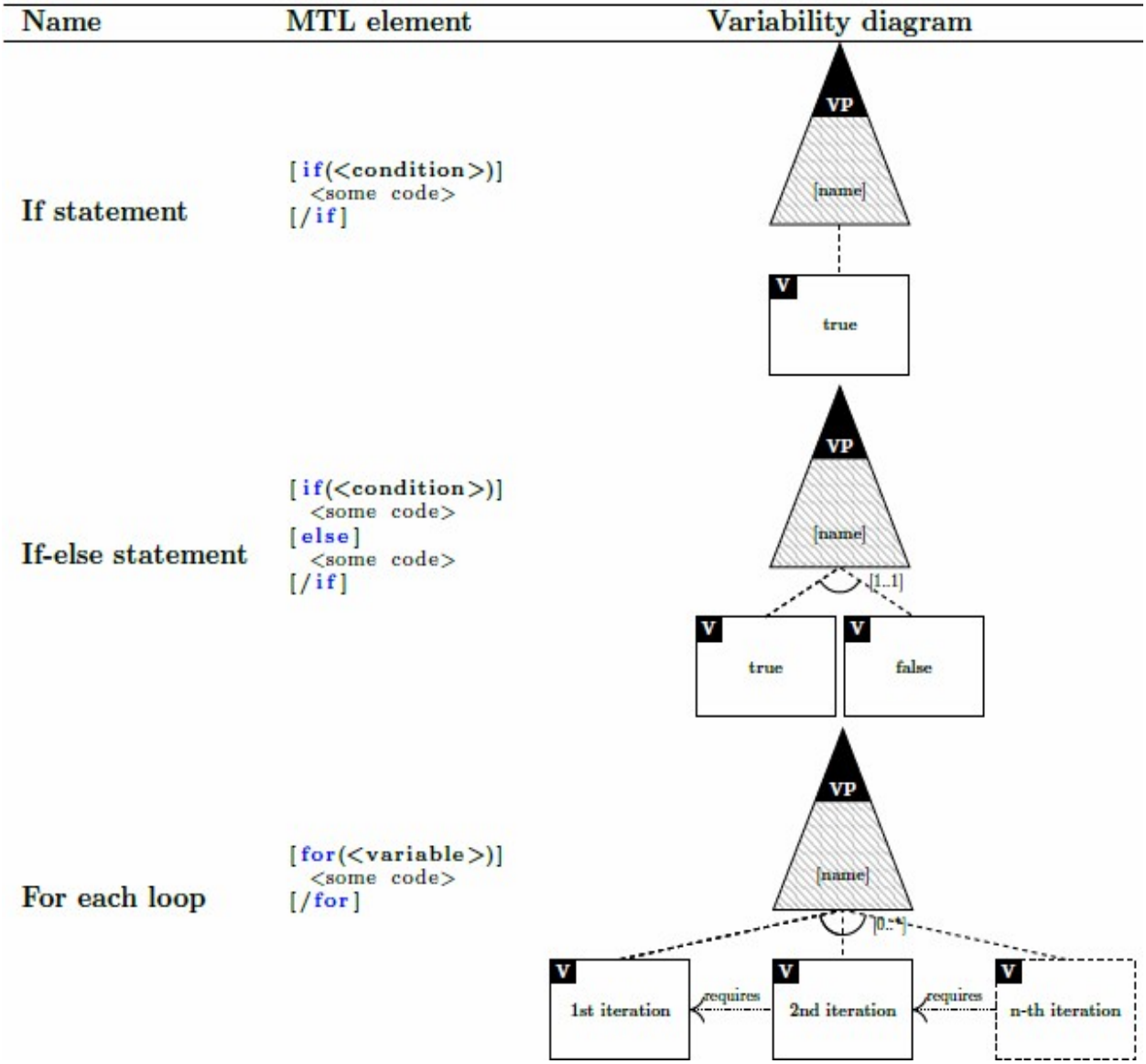

Рис. 2.7. Одностороннє відображення між метамовою та діаграмами мінливості

Щоб переконатися, що можна вибрати лише безперервний набір ітерацій, що починаються з першої ітерації, до відображення також слід додати деякі обмеження. Якщо вибрано (i)-й варіант ітерації, то також слід вибрати (i - 1)-й варіант ітерації. Отже, між цими варіантами має існувати вимога. Ця вимога є транзитивною, отже, якщо вибрано ітерацію, то всі попередні ітерації також вибрано.

Нульові ітерації можна змоделювати двома способами: додатковий варіант, який представляє відсутність ітерацій, або за допомогою альтернативного вибору, де мінімальна кратність дорівнює нулю. У цій дисертації обрано останнє, але обидва варіанти однаково правильні.

В цій роботі основна увага приділяється оператору if та циклу for-each. Отже, нескінченна кількість ітерацій підстановки не є проблемою і також не відображається на діаграмах. Кількість ітерацій у циклі for-each також не є проблемою, оскільки їх можна узагальнити і вони зображені на діаграмах пунктирними лініями. Повне відображення показано на рисунку 2.7.

Елементи MTL можуть бути вкладеними, це також має бути показано на діаграмі змінності. Вкладеність елементів може бути змодельована зв'язком вимог між варіантом і точкою варіації. Семантичне значення такої вимоги полягає в тому, що коли вибрано варіант, вибір має бути зроблено щодо точки варіації. Коли точка варіації вкладена в оператор if або if-else, тоді варіант може просто вимагати вкладеної точки варіації.

Це не так тривіально для циклу for-each. Через повторюваність циклу варіанти дублюються. Тому що для кожної ітерації вибір потрібно робити заново. Кожен вибір, який можна зробити, дублюється на кількість ітерацій. Оскільки кількість ітерацій не обмежена, кожен вибір також має дублюватися необмежену кількість разів. Це неможливо намалювати, тому використовується та сама стратегія, створюючи елементи змінності пунктирними лініями. Те, що моделі є необмеженими, не є проблемою для частини аналізу, оскільки точки варіації та варіанти можуть бути узагальнені.

У цій роботі передбачається, що замовник є безпосереднім автором введення шаблону. Це не так, якщо трансформація моделі виконується перед виконанням шаблону. Але навіть тоді для розробника шаблону всі зовнішні рішення все ще виходять із моделі. Тому всі цікаві рішення позначені як зовнішні.

2.3. Дослідження існуючих методик опрацювання проблеми варіативності в об’єктній мові

2.3.1. Динамічне завантаження

Коли мова об'єкта є об'єктно-орієнтованою мовою програмування, тоді можна вирішити проблеми мінливості в межах мови об'єкта. Одним із таких прикладів є шаблон стратегії [13]. Шаблон стратегії зображено на рисунку 2.2. У шаблоні стратегії можна вказати, як щось має оброблятися в коді. Це відображає вибір точки мінливості. Оскільки реалізація має відповідати інтерфейсу, її легко замінити.

Це може призвести до коду, який ніколи не використовується в певних конфігураціях, оскільки цілий клас або модуль потрібно включити або виключити. Це також потрібно в деяких випадках, коли потрібні деякі бібліотеки, але призводить до помилок компіляції, коли вони недоступні.

Щоб вирішити проблему з недоступністю бібліотек, можна реалізувати бібліотеку динамічного завантаження [32]. У цій техніці бібліотеки шукаються та завантажуються під час виконання. За допомогою динамічного завантаження можна створити програму з певним набором функцій. Системи плагінів часто використовують цю можливість.

Проблема з таким типом техніки полягає в тому, що вона передбачає додавання або відсутність функції. Більш детальні зміни у функціях неможливі. З такими типами технік єдина можлива модель - це список логічних значень. У модельно-орієнтованій розробці (MDE) модель часто є

більшою та складнішою, тому такі типи технік не можна використовувати в програмному проекті MDE.

2.3.1. Генеративні технології

Були проведені дослідження щодо моделювання мінливості в кодовій базі в контексті лінійок програмних продуктів (SPL). Одним із підходів є використання генеративної технології, такої як технологія фреймів. Це дає перевагу, що можна вносити більш детальні зміни в модулі або класи.

Різні реалізації технології фреймів порівнюються в [11]. Реалізація під назвою XML-based Variant Configuration Language (XVCL) [42] може бути використана в архітектурі SPL [42]. Наступником XVCL є Adaptive Reuse Technique (ART) [23]. Тоді як XVCL використовує мову на основі розширюваної мови розмітки (XML), яка є дуже багатослівною синтаксисом, ART використовує більше синтаксису с-препроцесора, який є менш багатослівним.

Проблеми з технологією фреймів полягають у тому, що робиться припущення, що модель є списком значень. Більш складні моделі даних, такі як моделі з асоціаціями, важко використовувати. Ще одним недоліком ART є те, що мова використовує синтаксис с-препроцесора для своїх if-структур і циклів. Таким чином, найнижчий можливий рівень деталізації - це один рядок, що може призвести до проблем, коли потрібна менша деталізація. Приклад показано в лістингу 2.13, з результатом у лістингу 2.14.

Лістинг 2.13. Елементарний приклад з ART

```
1 #set var_a = "AAA"
2 Value of var_a is: ?@var_a?
```

Лістинг 2.14. Результат прикладу ART

```
1 Value of var_a is: AAA
```

Ще одна генеративна техніка досліджується в [12]. У цій техніці кодова база розробляється шарами, як показано на рисунку 2.8. Кожен шар має ряд артефактів кодування. Шари можна складати, артефакти коду вищого рівня можуть уточнювати артефакти коду нижчого рівня. Уточнення може складатися із заміни або додавання (до або після). Це створено для моделі програмування, орієнтованої на функції [16].

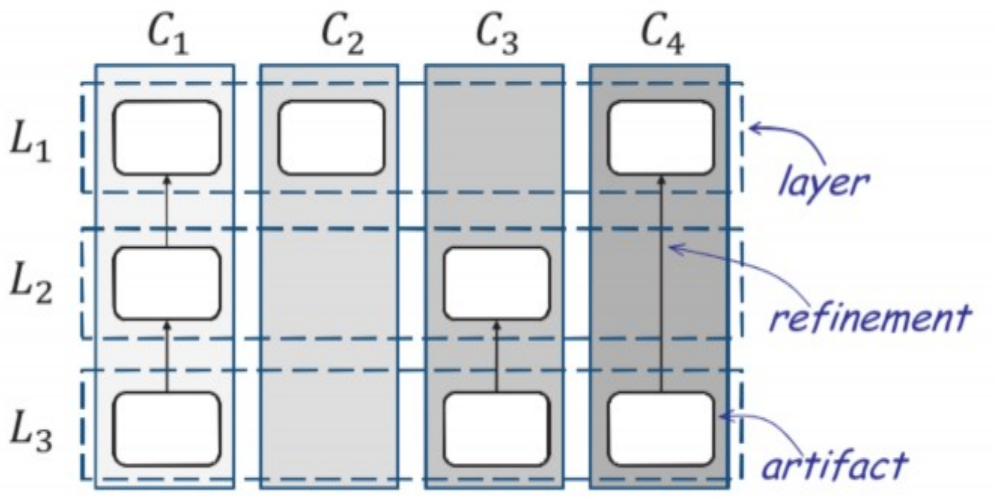

Рис. 2.8. Генеративна техніка на основі шарів [12]

Проблеми з цими генеративними підходами полягають у тому, що модель є дуже маленькою та дуже обмеженою порівняно з моделлю, яка використовується в процесі MDE. Моделі в цих генеративних підходах - це здебільшого список логічних значень або значень про те, які функції включені або виключені. У MDE ці моделі можуть бути набагато більшими та складнішими. Через використання кратності нуль або більше модель може бути навіть необмеженою. Також ці технології розроблені з наміром мати великий вплив на згенерований код.

2.3.3. Перевірка мови об'єкта

Інший підхід полягає у виявленні помилок, які можуть виникнути до виконання шаблону. Це вже можливо безпосередньо на згенерованому коді. Коли вводиться метамова, перевірка на наявність помилок стає складнішою

через дві переплетені мови. Були проведені дослідження для виявлення синтаксичних проблем мови об'єкта [5]. Щоб перевірити синтаксис, метамову та мову об'єкта слід об'єднати. За допомогою об'єднаної граматики можна перевірити синтаксис метамови та мови об'єкта.

Розпочато обговорення семантичної перевірки мови об'єкта мови об'єкта [4]. Що перевіряло б проблеми, обговорені в цьому розділі. В основному через неповну природу шаблонів неможливо виявити всі помилки. Деякі проблеми можуть бути помилкою залежно від вмісту вхідної моделі. В обговоренні ці можливі помилки позначені як попередження.

Рішення ще не запропоновано, тому це ще не можна використовувати. Це лише виявляло б проблеми, але не вирішувало їх.

Висновки до розділу

У другому розділі проведено ґрунтовний аналіз використання шаблонів як інструменту для генерації коду на основі моделей, із врахуванням їхньої варіативності.

Досліджено типові проблеми, що виникають при застосуванні шаблонів, зокрема неявну розсіяну умову та надлишковий об’єктний код, які негативно впливають на якість розробки програмного забезпечення та його подальшу підтримку. Вивчено залежності в об’єктно-орієнтованих мовах програмування та концепцію Software Product Line (SPL), яка забезпечує систематичне управління варіативністю в розробці програмних продуктів.

Особлива увага приділена моделюванню варіативності та мінливості в шаблонах, що дозволяє ефективніше адаптувати програмне забезпечення до специфічних вимог. Проаналізовано існуючі методики вирішення проблем варіативності, зокрема використання динамічного завантаження, генеративних технологій та перевірки мови об’єкта.

Встановлено, що генеративні технології дозволяють автоматизувати процеси розробки з урахуванням варіативності, тоді як динамічне

завантаження сприяє підвищенню гнучкості програмних систем. Отримані результати підкреслюють необхідність подальшого вдосконалення підходів до управління варіативністю для забезпечення більшої ефективності та надійності процесів розробки програмного забезпечення.

РОЗДІЛ 3. МОДЕЛІ, МЕТОДИ ТА ПІДХОДИ ПОКРАЩЕННЯ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ ШАБЛОНІВ

3.1. Представлення підходів до рішення проблеми варіативності в шаблонах

Для подальшого аналізу проблеми PS1 та PS2, що описані в другому розділі створюються діаграми мінливості. Ці діаграми описують мінливість шаблонів. Вони покажуть, що метамова не моделює відношення, які існують у мові об'єкта.

3.1.1. Рішення проблеми явної розсіяної умови

Можна показати діаграму мінливості прикладу в лістингу 2.1. На рисунку 3.1 оригінальний приклад показано праворуч. Можна виразити цей шаблон у діаграмі мінливості.

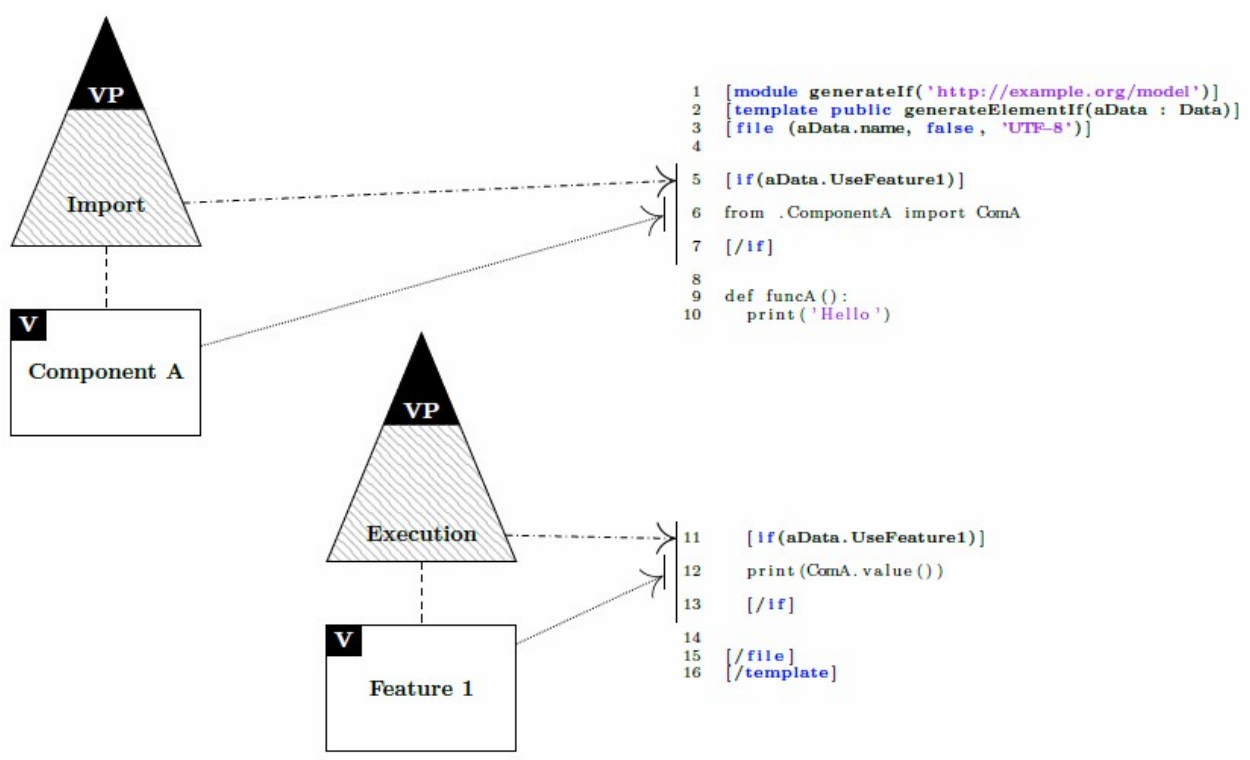

Рис. 3.1. Процес мінливості в PS1

У шаблоні є дві if-структури в метакоді: перша з рядка 5 по 7 і друга з рядка 11 по 13. Для кожної if-структури на діаграмі показано точку мінливості, в даному випадку точку мінливості Import та точку мінливості Execution відповідно. Для кожної точки мінливості в кодовій базі показано додаткову залежність. Оскільки обидва if не мають частини else, для кожної if-структури показано лише один необов'язковий варіант: Component A та Feature 1. Вони також мають залежність артефакту від тіла if-структур. Що призводить до діаграми, зображеної на лівій стороні рисунка 3.1.

В оригінальному визначенні проблеми з другого розділу визначено зв'язок між різними частинами об'єктного коду. У цьому прикладі це відношення use-def між рядком 12 (використання) і рядком 6 (визначення). Цей зв'язок не показано в моделі мінливості. Відношення use-def між двома частинами об'єктного коду явно не визначено в метакоді, хоча воно все ще існує в об'єктному коді.

3.1.2. Рішення проблеми неявної розсіяної умови

PS2 - це більш складне узагальнення PS1. У PS2 умова безпосередньо не відображається в іншій частині шаблону. У прикладі використовується неявний цикл і підшаблон.

Лістинг 3.1. PS2 - Неявна клонована умова без підшаблону та неявного циклу

```
1  [module generateImplicitIf('http://example.org/model')]
2
3  [template public generateElementImplicitIf(aData : Data)]
4    [file (aData.name, false, 'UTF-8')]
5      [if(aData.UseFeature1 && aData.B->notEmpty() && aData.B.any(b | b.B_value > 0))]
6      from .ComponentA import ComA
7      [/if]
8      def funcA():
9        print("Hello")
10       [if(aData.UseFeature1)]
11       print("world")
12       [for(singleInstanceB : DataB | aData.B)]
13       [if(singleInstanceB.B_value > 0)]
14       print(ComA.Calculate([singleInstanceB.B_value /]))
15       [/if]
16       [/for]
17       [/if]
18   [/file]
19 [/template]
```

У лістингу 3.1 вони видалені та замінені еквівалентними елементами з мови Model to Text Language (MTL). Це спрощує створення діаграми мінливості. Відношення use-def все ще виділено в рядках 6 і 14.

Єдина відмінність від діаграми мінливості PS1 полягає в тому, що в PS2 є неявний цикл for. Діаграма мінливості PS2 показана на рисунку 3.2.

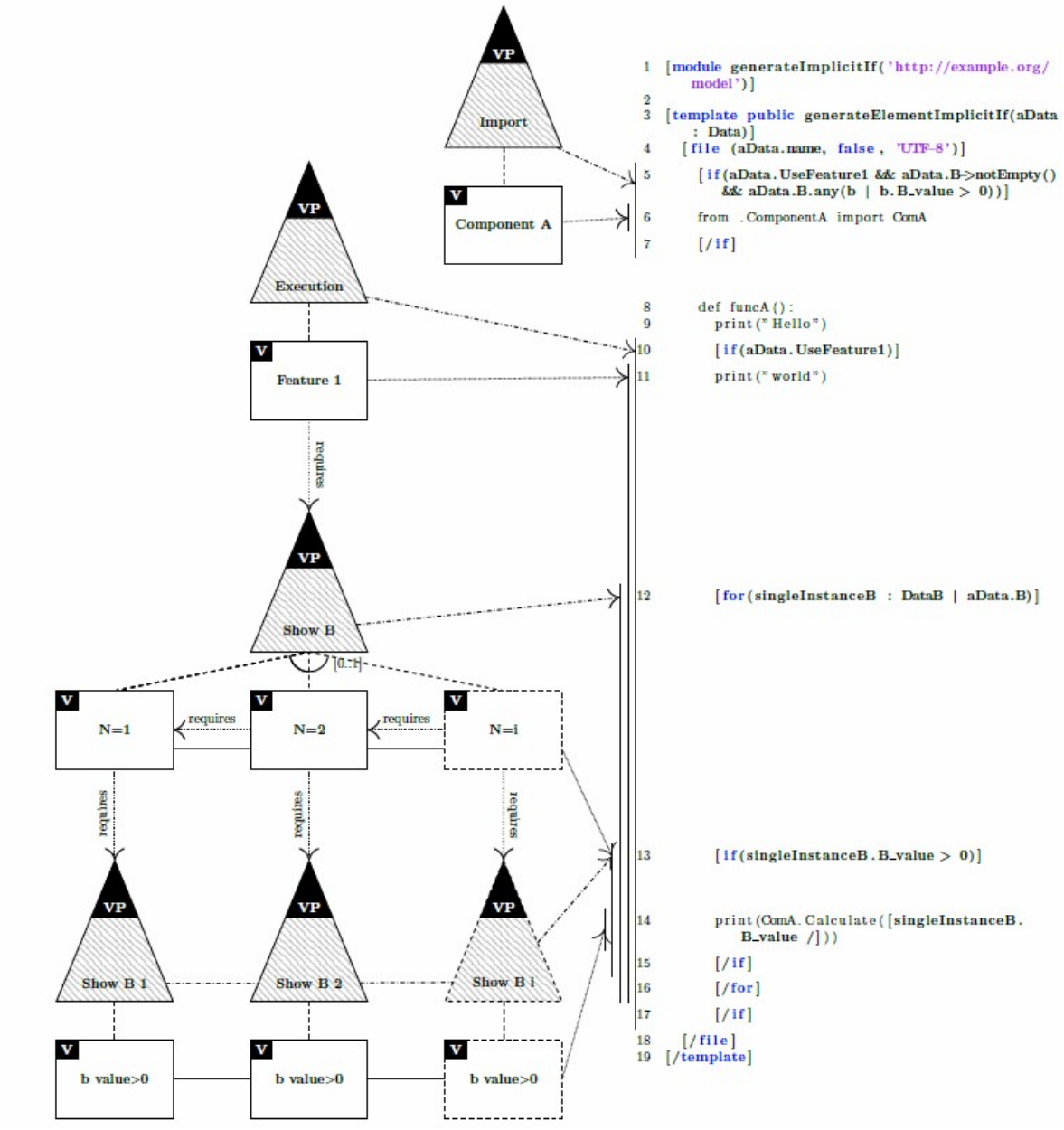

Рис. 3.2. Процес мінливості в PS2

Для if-структури в рядку 5 на діаграмі показано точку мінливості та варіант, відповідно Import та Component A. Для if-структури в рядку 10 також

можна застосувати цю саму процедуру, є точка мінливості Execution та варіант Feature 1.

Для циклу for також має бути показано точку мінливості під назвою Show B на рисунку 3.2. Для кожної кількості ітерацій має бути показано варіант під назвою N=1, N=2, N=3 тощо. Це нескінченна кількість варіантів, оскільки немає обмежень на список aData.b у моделі. Щоб все ще мати можливість намалювати цю модель, у третьому варіанті використовується пунктирна лінія, що вказує на нескінченну кількість варіантів, а змінна і використовується як кількість ітерацій.

Для останньої найбільш вкладеної if-структури на діаграмі слід показати точку мінливості та необов'язковий варіант. Оскільки цей оператор if вкладено в цикл for, для кожної можливої ітерації слід показати точку мінливості та варіант. Це знову нескінченна кількість точок мінливості та варіантів, тому остання точка мінливості та варіант намальовані пунктирною лінією, щоб вказати на нескінченну кількість варіантів та точок мінливості.

Найнижчі варіанти (під назвою bvalue > 0) вказані в рядку 14, цей рядок є стороною використання відношення use-def, обговореного в другому розділі. Цю залежність можна змоделювати як requires. Ці requires не показані, оскільки вони не існують в оригінальному метакоді. Отже, цей зв'язок між двома елементами явно не визначено в метакоді, хоча він все ще існує в об'єктному коді.

3.2. Моделювання відношення вимог на основі діаграм мінливості

На діаграмах мінливості PS1 та PS2 не існує вимоги (requires), хоча все ще існує зв'язок між фрагментами об'єктного коду, але не в метакоді. Ці відношення в обох випадках є відношенням використання-визначення (use-def), але можуть бути й іншими типами відношень. Отже, у цьому розділі ми намагаємося змоделювати це відношення вимоги з діаграми мінливості новою мовою, яка називається Intermediate Variant Language (IVL). Також

перевіряється, які відношення та об'єкти також слід моделювати мовою, щоб вимога працювала правильно.

Щоб змоделювати вимогу в шаблоні, потрібне більш детальне розуміння метамоделі діаграм мінливості. Метамодель показана на рисунку 3.3. Ця метамодель показує всі частини мови мінливості. Лише не всі частини потрібні для моделювання необхідного відношення. Існує аргумент для моделювання кожної частини метамоделі в шаблоні. Щоб зменшити складність і в дусі мінімалістичної метамови [5, 29], вибрано лише включити наймінімальнішу частину метамоделі.

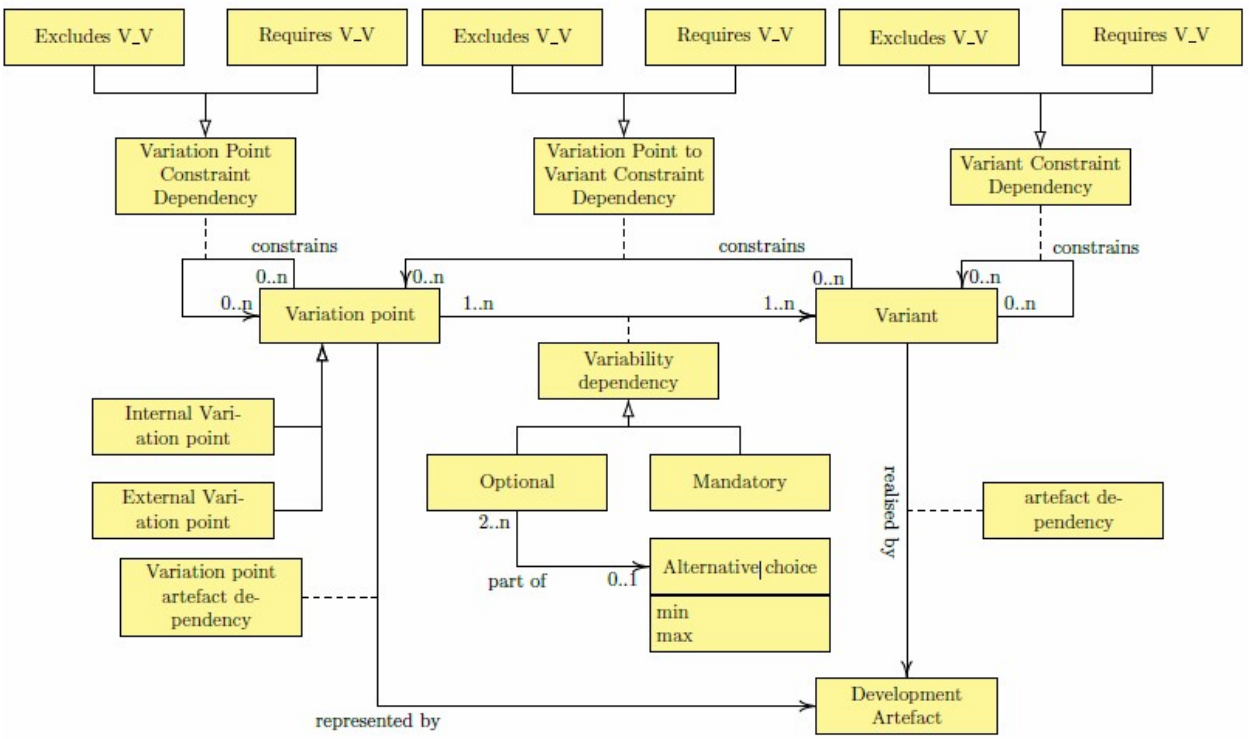

Рис. 3.3. Метамодель діаграми мінливості

Для IVL необхідні лише два елементи: варіанти та вимоги, які визначаються наступним чином:

- Визначити варіант: <!variant [name] !>[object code|requires|variant] <!/variant!> Визначає варіант з іменем, вкладеним об'єктним кодом і вкладеними вимогами. Також варіанти можуть бути вкладені один в одного. Результат варіанта може бути одним із двох: він повністю видаляється,

включаючи вкладений код (виключено), або вкладений код зберігається, а визначення видаляється (включено).

- Вимагати варіант: <!requires [name] /!> Вимагає, щоб варіант, вказаний з іменем, був включений.

Відношення, якого бракує на діаграмах мінливості, - це відношення вимоги, зокрема клас Requires V_V. Щоб змоделювати цей клас, також слід змоделювати відношення до інших класів. Клас Requires V_V має 2 відношення до варіантів, тому обидва варіанти слід моделювати. Один варіант є ціллю, а інший - джерелом. Щоб змоделювати джерело, вимога вкладена у варіант, так що вона потрібна, коли джерело включено. Щоб змоделювати цільове відношення, використовується ім'я. Це ім'я вказує на варіант. Це також вимагає, щоб варіант був явно визначений мовою. Це не так у випадку з вихідним варіантом, який може бути явно названим варіантом, але також може бути неявним варіантом, визначеним оригінальною метамовою, такою як MTL.

Отже, з змодельованою вимогою, потрібно змоделювати лише варіант. Варіанти за визначенням не мають прямих властивостей, але моделюючи вимоги, необхідно дати варіантам ім'я. Крім того, варіанти мають ряд відношень. Деякі з цих відношень слід моделювати. Варіант має відношення до артефакту розробки, у випадку шаблонів це фрагмент об'єктного коду. Отже, щоб змоделювати це, фрагмент об'єктного коду можна вкласти у варіант. Варіанти також мають відношення вимоги та відношення виключення до інших варіантів. Відношення вимоги можна змоделювати, вклавши вимогу всередину варіанта. Відношення виключення можна виключити з моделювання, оскільки решта все ще може бути дійсною без цього відношення. Отже, в дусі мінімалістичної мови це відношення не моделюється.

Варіанти також мають два різних відношення до точок мінливості. Залежність мінливості - це той факт, що варіант належить до однієї або кількох точок мінливості, а точка мінливості має один або кілька варіантів.

Це пов'язано з тим, що точки мінливості необхідні для визначення місця внесення змін. У текстовій мові це можна змоделювати простіше та інтуїтивніше. Оскільки розташування варіанта також може бути точкою мінливості.

Інше відношення між варіантами та точками мінливості - це відношення обмеження, яке існує двох різних типів: Requires та Excludes. Відношення виключення не моделюється, оскільки модель все ще може бути дійсною щодо метамоделі без таких відношень. Інше - це вимога між варіантом і точкою мінливості. Щоб змоделювати це, потрібно вивчити семантику. Коли варіант вимагає точку мінливості, він визначає, що вибір має бути зроблений у точці мінливості, коли вибрано цей варіант [32]. Оскільки розташування варіанта є точкою мінливості, це означає, що якщо точка мінливості потрібна, точка мінливості має бути вкладена у варіант. Отже, цей тип відносин можна моделювати, вкладаючи варіанти. Отже, два відношення між варіантом і точкою мінливості вже можна моделювати, тому точку мінливості не потрібно моделювати явно. Вона все ще існує, лише для кожного варіанта існує неявна точка мінливості.

У наступному розділі обговорюється, як цю мову можна обробити, щоб вивести компілювальний об'єктний код, але спочатку показано застосування цієї мови до PS1, PS2 та PS3.

3.2.1. Вирішення проблеми клонованої умови

Елементи мови можна застосувати до PS1. Результат показано в лістингу 3.2. У рядках з 5 по 7 визначено варіант. Це те саме місце, що й оригінально скопійована умова. Щоб переконатися, що варіант включено, потрібно вказати вимогу. Це зображено в рядку 12. Коли вимога виконується, вона гарантує, що варіант з іменем ComA включено.

Застосування варіантів у PS2 показано в лістингу 3.3. У рядках з 5 по 7 визначено варіант. Цей варіант має бути потрібним у шаблоні, коли це необхідно. Вимога розташована близько до сторони використання

відношення використання-визначення (use-def). Вимога вказана в рядку 19, трохи вище використання ComA.

Лістинг 3.2. PS1 – Явна клонована умова, застосована з проміжною варіантною мовою

```
1  [module generateIf('http://example.org/model')]
2  [template public generateElementIf(aData : Data)]
3  [file (aData.name, false, 'UTF-8')]
4
5  <!variant ComA!>
6  from .ComponentA import ComA
7  <!/variant!>
8
9  def funcA():
10   print('Hello')
11   [if(aData.UseFeature1)]
12   <!requires ComA /!>
13   print(ComA.value())
14   [/if]
15
16 [/file]
17 [/template]
```

Лістинг 3.3. PS2 - Неявна клонована умова, застосована з проміжною мовою варіантів

```
1  [module generateImplicitIf('http://example.org/model')]
2
3  [template public generateElementImplicitIf(aData : Data)]
4    [file (aData.name, false, 'UTF-8')]
5      <!variant ComA!>
6      from .ComponentA import ComA
7      <!/variant!>
8      def funcA():
9        print("Hello")
10       [if(aData.UseFeature1)]
11       print("world")
12       [showB(aData.b) /]
13       [/if]
14   [/file]
15 [/template]
16
17 [template public showB(singleInstanceB : DataB)]
18   [if(singleInstanceB.b_value > 0)]
19   <!requires ComA /!>
20   print(ComA.Calculate([singleInstanceB.b_value /]))
21   [/if]
22 [/template]
```

Зверніть увагу, що в обох прикладах умови, вказані в оригінальній ситуації, видалені, тому їх також не потрібно підтримувати. Коли варіант включено, він не керується розробником, це робить механізм шаблонів. Отже, коли умова для включення варіанта змінюється, інший код не потрібно змінювати.

3.2.2. Надлишковий об'єктний код з варіативною мовою

Тепер розглянемо проблему PS3, яка описана в другому розділі. У цій задачі фрагменти об’єкта зайві через помилку розробника. Кілька фрагментів повторюваного об’єктного коду можна включити за умов, які не обов’язково повинні бути взаємовиключними. Це більше не аналізується, тому що ця проблема виникає через те, що стан стає важчим. Лише рішення, представлене в цьому розділі, повністю видаляє клоновані умови. Таким чином, в умові не може бути помилок. Оскільки кожен варіант можна включити лише один раз, кілька фрагментів об’єктного коду можуть вимагати того самого варіанту без створення помилок. Результат показано в лістингу 3.4.

Лістинг 3.4. PS3 – Надлишковий об’єктний код, застосований із проміжною варіантною мовою

```
[module generateRedundant('http://example.org/model')]

[template public generateElementRedundant(aData : Data)]
[file (aData.name, false, 'UTF-8')]

<!variant ComA!>
from .ComponentA import ComA
<!/variant!>

def funcA():
  print("Hello")
  [if(aData.UseFeature1)]
  print(ComA.value())
  <!requires ComA /!>
  [/if]
  [if(aData.UseFeature2)]
  print(ComA.value2())
  <!requires ComA /!>
  [/if]

[/file]
[/template]
```

3.3. Стратегія обробки шаблонів

Спеціалізовану мову представлено в розділі 3.2, і її потрібно обробити. Ця мова має варіанти та вимоги, як обговорювалося. Коли потрібно обробити вимогу, варіант має бути включено. Можна включити це в мову шаблонів, наприклад MTL. Це призведе до складнішої мови шаблонів. У цій роботі

обрано дослідження окремої мови, яка обробляється після виконання механізму шаблонів.

Аргументом для обробки IVL є те, що мову та механізм шаблонів не потрібно адаптувати для цієї мови. Тоді IVL сумісний з будь-яким механізмом шаблонів. Також механізм шаблонів вирішує всі проблеми, пов'язані з неявними варіантами, тому залишаються лише явні варіанти. Неявний варіант все ще може вимагати явного варіанта, але ці вимоги проходять через весь файл, згенерований механізмом шаблонів.

У лістингах 3.5 та 3.6 показано два можливих виходи механізму шаблонів, які можна створити за допомогою шаблону в лістингу 3.2. Лістинг 3.7 показує можливий результат механізму шаблонів для шаблону в лістингу 3.3.

Лістинг 3.5. PS1 – мова проміжного варіанту, де MTL вже оброблено і де aData.UseFeature1 = = true

```
<!variant ComA!>
from .ComponentA import ComA
<!/variant!>

def funcA():
  print('Hello')
  <!requires ComA /!>
  print(ComA.value())
```

Лістинг 3.6. PS1 – мова проміжного варіанту, де MTL вже оброблено та де aData.UseFeature1 == false

```
<!variant ComA!>
from .ComponentA import ComA
<!/variant!>

def funcA():
  print('Hello')
```

Лістинг 3.7. PS2 – мова проміжного варіанту, де MTL вже оброблено

```
<!variant ComA!>
from .ComponentA import ComA
<!/variant!>

<!requires ComA /!>
print(ComA.Calculate(10))
<!requires ComA /!>
print(ComA.Calculate(15))
```

Ці приклади дуже малі і не потребують складного алгоритму для обробки. Якщо вимога вказана у файлі, то варіант має бути включено, інакше його слід виключити.

Цей алгоритм не такий тривіальний для складніших прикладів. Наприклад, у лістингу 3.8 вимога вкладена не лише в один варіант, а в 2 варіанти. Отже, для цього прикладу спочатку слід обробити вимоги до варіанту A (рядок 7) та B (рядок 8), після чого слід обробити вимогу до варіанту C (рядок 3). Це повторюваний процес, і його можна повторювати частіше, ніж два кроки в цьому прикладі. Ще одне важливе питання з цим прикладом полягає в тому, що варіанти A та B мають бути включені для того, щоб можна було обробити вимогу до варіанту C. Це свого роду логічна кон'юнкція.

Лістинг 3.8. Більш складний приклад проміжної мови

```
1  <!variant A!>
2    <!variant B!>
3      <!requires C /!>
4    <!/variant!>
5  <!/variant!>
6
7  <!requires A /!>
8  <!requires B /!>
9  <!variant C!>
10   /* some code */
11 <!/variant!>
```

Щоб обробити також складніші екземпляри цієї проміжної мови, представлено два різних алгоритми. Один дуже простий, але потенційно повільний через кількість проходів по кодовій базі. Другий складніший у реалізації, але вимагає лише двох проходів по коду.

3.3.1. Простий багатопрохідний алгоритм

Псевдокод для першого алгоритму вказано в алгоритмі 1. У цьому алгоритмі всі вимоги, які не вкладені у варіант, обробляються шляхом пошуку вимог, а потім обробки всіх варіантів, які вказують вимоги. Це повторюється, доки більше нічого не зміниться, буде отримано фіксовану

точку. Після чого всі необроблені варіанти видаляються. Якщо кожна вимога та варіант видалені, то залишається лише об'єктний код.

Algorithm 1 Process single step in the intermediate language

```
1:  procedure PROCESS
2:      C ← true
3:      while C do                                  ▷ Loop until nothing changes
4:          C ← false
5:          Create empty set R
6:          for every require r do                  ▷ Find all requires
7:              if r not nested in a variant then
8:                  R ← R ∪ {r}
9:                  remove r from file
10:                 C ← true
11:             end if
12:         end for
13:         for every variant v do                  ▷ Process the variants
14:             if v ∈ R then
15:                 Replace variant v with content of v
16:             end if
17:         end for
18:     end while
19:     for every variant v do                      ▷ remove left over variants
20:         Remove variant v
21:     end for
22: end procedure
```

Цей алгоритм може призвести до великої кількості проходів. Щоб показати це, потрібно ввести трохи позначень. Щоб вимога була оброблена, має бути вже вимагано кілька варіантів. Ці варіанти - це ті, в яких вкладено вимоги. Це можна представити множиною, оскільки дублікати не мають додаткового семантичного значення. Вимога визначає лише один варіант. Вимогу можна позначити як {V1; V2; … ; Vi} → Vt, де V1 до Vi - це варіанти, в яких вкладено вимогу, а Vt - цільовий варіант. Зауважте, що {} → Vi можливо, коли вимога не вкладена в жодний варіант.

Нехай є кодова база з n кількістю варіантів, V1, V2, ... , Vn. Також є вимога для кожного варіанта Vi у вигляді {Vi-1} → Vi, крім першого. Для першого є вимога наступного вигляду {} → V1. Приклад, де n = 4, показано в лістингу 3.9. Якщо алгоритм 1 запускається на такому файлі, він виконує 2n проходів по файлу. Це пов'язано з тим, що він знаходить лише одну вимогу

за ітерацію, тоді як йому все ще потрібно два повних проходи за ітерацію. Через велику кількість проходів він потенційно може бути повільним.

Лістинг 3.9. Приклад повільної роботи з простим алгоритмом

```
<!requires V1 /!>

<!variant V1!>
  <!requires V2 /!>
<!/variant!>

<!variant V2!>
  <!requires V3 /!>
<!/variant!>

<!variant V3!>
  <!requires V4 /!>
<!/variant!>

<!variant V4!>
  /* some code */
<!/variant!>
```

3.3.2. Двопрохідний алгоритм

Покращенням попереднього алгоритму є те, що наступний алгоритм потребує менше проходів по коду. У попередньому алгоритмі потрібно 2n проходів. Тепер буде представлено другий алгоритм, який потребує лише двох проходів.

Другий алгоритм складається з трьох частин. У першій частині вся інформація агрегується в графі. Після чого в другій частині ця інформація аналізується за допомогою модифікованого алгоритму пошуку в ширину. Друга частина призводить до набору варіантів, які потрібні. З цим набором варіанти в коді обробляються в останній частині. Основною перевагою цього алгоритму є те, що він потребує лише двох проходів: один у першій частині та один у третій частині.

Цей алгоритм базується на графі, де кожен варіант і вимога є вузлами. Можливо, більш інтуїтивним способом є моделювання вимог як ребер, а варіантів як вузлів. Такий граф недостатньо потужний, щоб виразити кожен випадок, можливий в IVL. Наприклад, у лістингу 3.8 необхідно, щоб варіант A та варіант B були включені, перш ніж можна буде включити варіант C. Коли вимога представлена ребром, необхідно, щоб ребро походило від A та

від B, що неможливо. Через це та сценарій необхідно, щоб вимога також моделювалася як вузли, які мають лише одне вихідне ребро, але кілька вхідних ребер.

Граф створюється в алгоритмі 2. Якщо вузол варіанта для певного варіанта використовується, але ще не існує, він створюється автоматично. Для кожної вимоги r = {V1; V2; :::; Vi} → Vt у графі створюється вузол. Існує спрямоване ребро від вузла вимоги до цільового варіанта. Також є спрямовані ребра від варіантів, які потрібні, до вимог. Щоб спростити етап аналізу, для кожного вузла вимоги також є ребро від спеціального початкового вузла S до вимоги. Цей вузол використовується для етапу аналізу, щоб розпочати обробку всіх вимог, але його також можна розглядати як моделювання всіх неявних варіантів з мови шаблонів.

Algorithm 2 Create the variant and requires graph

```
1: procedure CreateGraph                          ▷ creation of the graph
2:     Create empty directed graph G
3:     Add start node S
4:     for every require r do
5:         Add a require node r
6:         v_specified is the variant specified by r
7:         Add v_specified if not yet added to G
8:         Add an edge from r to v_specified
9:         for every variant v that contains r do
10:            Add v if not yet added to G
11:            Add an edge from v to r
12:        end for
13:        Add an edge from S to r
14:    end for
15:    Return G
16: end procedure
```

Для аналізу цієї інформації використовується адаптований пошук у ширину (BFS). Зазвичай в алгоритмі BFS кожне вихідне ребро додається до черги, коли вузол обробляється. Це все ще відбувається для вузлів варіантів, але для вузлів вимог це неможливо. Для кожного вузла вимоги зберігається набір, щоб запам'ятати, які з вхідних ребер вже оброблені. Якщо всі вхідні ребра оброблені, то вихідне ребро додається до черги. Це робиться для того,

щоб переконатися, що всі варіанти, які знаходяться навколо вимоги, також включені.

Якщо вузол варіанта обробляється в алгоритмі, то він також додається до результуючого набору. Це робиться для того, щоб алгоритм не обробляв вузол варіанта двічі. Оскільки набір варіантів є скінченним, а алгоритм не обробляє варіанти двічі, він має завершитися. Результуючий набір також використовується в третій і останній частині алгоритму. Крок аналізу зображено в алгоритмі 3.

Algorithm 3 Find all active variants

```
1: procedure FindActiveVariants(G, S)                    ▷ processes the graph
2:     For every require r in G create an empty set E_r
3:     Create empty set R
4:     Create empty queue Q
5:     for each edge e that is outgoing from S do
6:         enqueue(Q, e)
7:     end for
8:     while Q is not empty do
9:         e ← dequeue(Q)
10:        n ← pointsTo(e)
11:        if n is not in R then
12:            if n is a variant node then
13:                v ← n
14:                add v to R
15:                for each edge e that is outgoing from v do
16:                    enqueue(Q, e)
17:                end for
18:            else                                      ▷ n is a require node
19:                r ← n
20:                Add e to E_r
21:                T ← all the incoming edges to n
22:                if T = S_r then
23:                    enqueue(Q, outgoing edge from n)
24:                end if
25:            end if
26:        end if
27:    end while
28:    Return R
29: end procedure
```

Коли результуючий набір відомий, тоді потрібно обробити лише файли з варіантами. Коли варіант, що обробляється, знаходиться в результуючому

наборі, вміст зберігається, а варіант видаляється. Якщо варіант не знаходиться в результуючому наборі, то вміст видаляється разом з варіантом. Ця частина показана в алгоритмі 4.

Algorithm 4 Process variant file so that it generates a resulting file

```
1: procedure ProcessFile(R)                          ▷ Process a file
2:     for every variant v do
3:         if v in R then
4:             Replace variant v with content of v
5:         else
6:             Remove v
7:         end if
8:     end for
9: end procedure
```

Цей алгоритм має лише два проходи по коду. Кількість проходів цього алгоритму не залежить від кількості варіантів або вимог у коді. Тоді як попередній алгоритм потребував можливо 2n проходів по кодовій базі. У цьому розділі дано відповіді на дослідницькі питання 2 та 3, що подані в першому розділі.

Отже, аналізуючи проблеми PS1 та PS2, створюється діаграма мінливості з шаблонів. Ці діаграми мінливості відповідають метамоделі діаграми мінливості. На цих діаграмах шаблонів відсутня ключова частина - вимога. Вимога моделюється в метамоделі діаграми мінливості, але не моделюється в метамові MTL.

Таким чином, це моделюється в другій метамові, яка обробляється після оригінальної метамови. Можна створити простий алгоритм для обробки другої метамови.

Цей алгоритм має лише один недолік: кількість проходів може залежати від кількості варіантів. Представлено другий алгоритм, який агрегує всю інформацію, аналізує інформацію та в кінці застосовує цю інформацію. Цей алгоритм потребує лише двох проходів по коду.

3.4. Структура згенерованого файлу

У лістингах 2.5 та 2.6 показана проблема PS4 на прикладі. У цій проблемі правила угоди переплетені з іншими аспектами, що може призвести до проблем із повторним використанням та обслуговуванням.

Наразі цей процес виглядає наступним чином. Спочатку виконується ряд перетворень "Модель в Модель", після чого модель стає достатньо простою для виконання перетворення "Модель в Текст". Це останнє перетворення призведе до компілювального коду. Остання частина процесу після перетворень "Модель в Модель" зображена на рисунку 3.4.

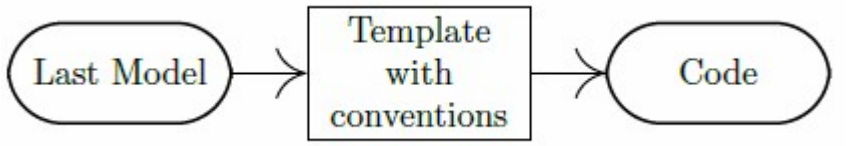

Рис. 3.4. Процес, який буде використовуватися з шаблоном, що відповідає угоді

Правила угоди, які реалізовані в шаблоні, можуть охоплювати різні аспекти того, як має бути написаний вихідний код. У випадку лістингів 2.5 та 2.6 очевидним правилом угоди є коментарі: заголовок у верхній частині файлу, заголовки розділів (імпорт, визначення функції та виконання) та заголовок функцій. Таке правило угоди існує не лише в цих прикладах, але й визначено в загальнодоступних посібниках зі стилю, які визначають правила угоди [14].

Інші правила угоди також можуть впливати на порядок елементів вихідного коду. Наприклад, весь імпорт має бути перед усім іншим у файлі вихідного коду, або класи мають бути визначені перед функціями. Інші правила угоди можуть визначати максимальну довжину рядка [35], кодування файлу [14] або навіть визначати додаткову мову для документації

[43]. У цьому розділі основна увага приділяється додаванню коментарів у файл вихідного коду.

У цьому розділі даються відповіді на питання 2: Чи можна змоделювати цю структуру та ці відношення в мові шаблонів, і 3: Як можна вирішити модель, щоб вона розв'язувалася в оригінальній цільовій мові.

У цьому розділі обговорюються гарні принтери та Lint. Це дві техніки, які можна використовувати для вирішення частини проблеми. Ці методи можна використовувати після того, як код буде згенеровано.

Гарні принтери можуть обробляти файл вихідного коду таким чином, щоб він відповідав стандартам форматування коду [7], що є частиною угод про кодування. Прикладами правил, які він може накладати на файл вихідного коду, є те, що відступ файлу правильний [19]. Лише гарний принтер не додає код, за винятком нових рядків і пробілів. Отже, існують обмеження щодо того, що може зробити гарний принтер. Наприклад, в оригінальному прикладі з лістингів 2.5 та 2.6 необхідно, щоб коментарі були присутні в результуючій кодовій базі. Гарний принтер не може додати їх до кодової бази, а якщо коментар відсутній, він не може його додати.

Lint - це спочатку програмне забезпечення, яке може перевіряти проблемні структури в коді C [31]. Наразі існує програмне забезпечення Lint для інших мов, таких як cpplint [15] для C++, Pylint [33] для Python та ESLint [12] для Javascript. Програмне забезпечення Lint може перевірити проект коду на наявність структур або стилів, які є незаконними за набором правил угоди. Програмне забезпечення можна використовувати для перевірки того, чи проект вихідного коду відповідає всім правилам угоди. Це ще неможливо зробити на об'єктному коді до процесу генерації. Нарешті, Lint лише перевіряє, чи допущені помилки, але не вимагає, щоб згенерований код відповідав стандарту.

Слід зазначити, що в проблемі PS4 переплітаються три різні аспекти: вміст, наданий моделлю, вміст, наданий шаблоном, і вміст, наданий правилами угоди. Вміст, наданий моделлю, використовується метамовою,

тоді як вміст, наданий шаблоном, є фіксованим об'єктним кодом. Вміст, наданий правилами угоди, може залежати від вмісту, наданого моделлю або шаблоном. У лістингу 3.10 показано фрагмент лістингу 2.5. Три різні аспекти видно у фрагменті. У рядку 18 частина def та частина () : мови об'єкта необхідні для забезпечення синтаксичної правильності результуючого коду. Тоді як частина [aData.Name /] в тому самому рядку - це вміст, наданий моделлю. Частина x та частина позиції x у рядку 21 - це вміст, наданий шаблоном, тоді як частина - - це вміст, наданий правилами угоди.

Лістинг 3.10. PS4 – фрагмент угод про кодування, приклад A (лістинг 2.5)

```
18 def [aData.Name /](x):
19   """Execute [aData.name /]
20   In:
21   x — x position
```

Лістинг 3.11. PS4 - Угоди про кодування, приклад A виконано

```
1  """
2  Description: -Generated-
3  Version: -Generated-
4  Authors: -Generated-
5  History: -Generated-
6  Date: 23-05
7  Copyright
8  """
9
10 #---------------------------
11 # imports
12 #---------------------------
13 from .ComponentA import ComA
14
15 #---------------------------
16 # function definition
17 #---------------------------
18 def Alpha(x):
19   """Execute Alpha
20   In:
21   x — x position
22
23   Out:
24   z — some description
25   """
26   print("Hello")
27   return ComA.calculate(x)
28
29 #---------------------------
30 # execution
31 #---------------------------
32 print(Alpha(321))
```

Щоб покращити якість вихідного коду, різні аспекти слід розділити. Це покращує зручність обслуговування та можливість повторного використання. Це загальне поняття, але також застосовується в цьому випадку до шаблонів.

Щоб далі визначити три аспекти, їх слід розділити. Аспект вмісту, наданого моделлю, вже визначено за допомогою метамови. Два інших аспекти ще не визначені. Поєднуючи вміст, наданий моделлю, та вміст, наданий шаблоном, можна витягти останній аспект, вміст, наданий правилами угоди, порівнюючи різні файли.

Лістинг 3.12. PS4 - Угоди про кодування, приклад B виконано

```
"""
Description: -Generated-
Version: -Generated-
Authors: -Generated-
History: -Generated-
Date: 23-05-
Copyright
"""

#---------------------------
# imports
#---------------------------
from .ComponentB import ComB

#---------------------------
# function definition
#---------------------------
def ExecBeta():
  """Execute ExecBeta
  In:
  --

  Out:
  z -- some description
  """
  return ComB.calculate(456)

def ExecGamma():
  """Execute ExecGamma
  In:
  --

  Out:
  z -- some description
  """
  return ComB.calculate(789)

#---------------------------
# execution
#---------------------------
print(Alpha(321))
```

Щоб об'єднати два з трьох аспектів, вміст, наданий моделлю, та вміст, наданий шаблоном, достатньо простого виконання механізму шаблонів. Результат показано в лістингах 3.11 та 3.12 для оригінального прикладу.

У лістингах 3.11 та 3.12 все ще показано два різні аспекти: вміст, наданий правилами угоди, та об'єднаний вміст з моделі та шаблону. Вміст, наданий правилами угоди, видно в прикладах, наприклад, у лістингу 3.11 у рядках з 1 по 12. Об'єднаний вміст з моделі та шаблону також видно, наприклад, оригінальний рядок print([aData.Name /]([aData.A value /])) призвів до рядка 32. Також можливо, що обидва аспекти знаходяться в одному рядку. Наприклад, у рядку 13 from та import взяті з граматики мови, але .ComponentA та ComA - це вміст із шаблону.

3.5. Процес отримання шаблону

Тепер, коли у файлах є лише два аспекти, можна побачити, що належить до вмісту, наданого правилами угоди, а що до нього не належить. Припускаючи, що два файли використовують одні й ті самі правила угоди та належать до однієї мови, можна побачити, які відмінності між двома файлами та припустити, які правила угоди для цих файлів. Отже, можна створити шаблон із цих спільних рис та відмінностей.

Лістинг 3.13. Шаблон загальної угоди

```
"""
Description: -Generated-
Version: -Generated-
Authors: -Generated-
History: -Generated-
Date: 23-05
Copyright
"""

#----------------------
# imports
#----------------------
[for(i : Import | File.imports)]
from [i.library /] import [i.class /]
[/for]

#----------------------
# function definition
#----------------------
[for(f : Function | File.functions)]
def [f.name /]([for(p : Parameter | f.parameter)][p.name /], [/for]):
  """Execute [f.name /]
  In:
  [for(p : Parameter | f.parameter)]
  [p.name /] -- [p.description /]
  [/for]

  Out:
  [if(f.return)][f.return.name /] -- [f.return.description /][/if]
  """
  [f.body /]
[/for]

#----------------------
# execution
#----------------------
[File.execution /]
```

У лістингу 3.13 показано загальний шаблон, який реалізує правила угоди. У цьому шаблоні передбачається, що тіло функцій розглядається як єдине ціле. Це вибір, щоб створити легше розуміння шаблону. У цьому шаблоні також можна розглядати кожен рядок або навіть меншу сутність. Це створило б набагато складніший шаблон, але збільшило б кількість правил, які він може застосовувати, наприклад, коментарі поруч зі змінною, визначеною в тілі.

Для використання шаблону угоди слід створити метамодель, яка описує вхідну модель. Метамодель показана на рисунку 3.5 і містить всю інформацію, яка використовується в шаблоні угоди. Зверніть увагу, що опис функції вказано, але він ніколи не використовується в шаблоні. Це може бути корисним, коли різні шаблони угод використовуються з тими самими моделями, так що в деяких угодах використовується більше інформації, а в інших - менше.

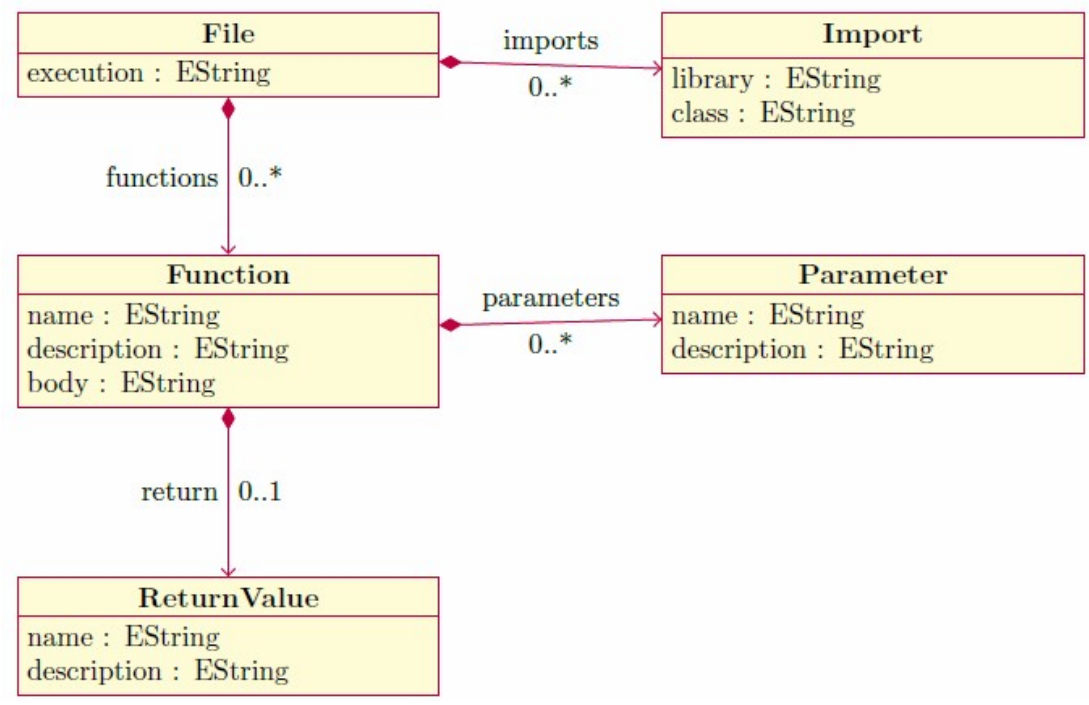

Рис. 3.5. Метамодель введення шаблону

Вхідну модель можна згенерувати за допомогою інструмента перетворення, який реалізує перетворення "Модель в Модель". Якщо це

зробити за допомогою інструмента перетворення "Модель в Модель", це призведе до дуже складного перетворення. Це тому, що багато коду є фіксованим і не потребує перетворення "Модель в Модель".

Інший варіант - представити вхідну модель як текстову мову. Це можливо за допомогою таких інструментів, як Xtext [45] та EMFText [10]. Якщо модель представлена текстом, для генерації моделі можна використовувати інструмент перетворення "Модель в Текст". Перевагою є те, що фіксований код легко вказати. Крім того, ті самі інструменти можна використовувати після останньої моделі, як це відбувається в поточній ситуації (див. рисунок 3.4).

Існують різні варіанти того, як представити модель у тексті. Один із варіантів - створити абсолютно нову мову. Інший варіант - використовувати вже відомі знання розробника. Приклад такого представлення показано в лістингу 3.14.

Лістинг 3.14. Текстова модель, схожа на Python

```
1  from .ComponentA import ComA
2
3  def Alpha(x):
4    print("Hello")
5    return ComA.calculate(x)
6
7  print(Alpha(321))
```

Цей приклад показує приклад представлення, дуже схожого на оригінальну мову, Python. Оскільки представлення дуже схоже на Python, воно майже не має кривої навчання, оскільки оригінальна мова вже відома розробнику, хоча вона все ще містить майже всю інформацію, необхідну для шаблону. Необхідно визначити відображення між представленням і метамоделлю.

Це відображення слід визначити для кожної мови, яка використовується. Слід зазначити, що не вся необхідна інформація присутня в цьому представленні. Це можна легко вирішити, додавши невеликі анотації до представлення. Анотації показані у виділеному фрагменті лістингу 4.6.

Лістинг 3.15. Текстова модель, схожа на Python, із додатковою інформацією

```
from .ComponentA import ComA

@description some text@
@return z, some value that needs a description@
def Alpha(x @description Lorem ipsum@):
  print("Hello")
  return ComA.calculate(x)

print(Alpha(231))
```

3.6. Застосування пропонованого підходу

Техніку, представлену в цьому розділі, можна використовувати з оригінальними прикладами. Оскільки це модель, представлена текстом, її можна згенерувати за допомогою метамови Model to Text Language (MTL), як обговорювалося раніше. Застосування цієї техніки застосовується до оригінального прикладу, а результат показано в лістингах 3.16 та 3.17.

Лістинг 3.16. PS4 – Правила кодування прикладу A виконано

```
from .ComponentA import ComA

@description some text@
@return z, some value that needs a
    description@
def [aData.name /](x @description Lorem
    ipsum@):
  print("Hello")
  return ComA.calculate(x)

print([aData.name/]([aData.a_value/]))
```

Лістинг 3.17. PS4 – Правила кодування прикладу B виконано

```
from .ComponentB import ComB

[for(b : DataB | aData.B)]
@return z, some description@
def Exec[b.B_value /]():
  return ComB.calculate([b.B_value /])
[/for]

print([aData.Name /]([aData.A_value /]))
```

Процес отримання правильного коду Python виглядає наступним чином. Спочатку механізм шаблонів запускається з оригінальною моделлю

та шаблонами в лістингах 3.16 або 3.17, що призводить до текстової моделі. Текстову модель можна використовувати під час виконання шаблону угоди, щоб отримати код Python, який відповідає правилам угоди. Цей процес показано як процес на рисунку 3.6.

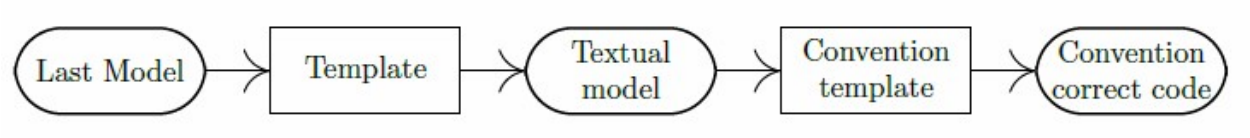

Рис. 3.6. Процес, який використовується з шаблоном, що відповідає умовам

У цьому випадку до коду додаються лише коментарі. Цей метод можна додатково вдосконалити, визначивши тіло функції як складну сутність, а не як єдину властивість. Тоді в тілі можна було б застосувати різні правила угоди, наприклад, щоб змінні визначалися на початку функції.

Отже, в оригінальних шаблонах переплітаються три різні аспекти. Ці три аспекти - це вміст, наданий моделлю, вміст, наданий шаблоном, і вміст, наданий угодою. Вони розділені шляхом видалення вмісту, наданого моделлю, і шаблон створюється з правил угоди. Щоб використовувати шаблон, створюється модель, яку можна представити текстом, подібним до оригінальної мови. Це представлення анотовано деякою додатковою інформацією для розміщення додаткової інформації про сутності. Нарешті, цей метод застосовується до оригінальних прикладів.

Висновки до розділу

У третьому розділі розглянуто моделі, методи та підходи, спрямовані на підвищення ефективності використання шаблонів у процесі розробки програмного забезпечення. Представлено різноманітні рішення проблем

варіативності в шаблонах, зокрема явної та неявної розсіяної умови, які сприяють зменшенню складності та підвищенню читабельності коду. Особливу увагу приділено моделюванню відношень вимог із використанням діаграм мінливості, що дозволяє систематизувати підходи до управління варіативністю та розв’язувати проблему клонованих умов.

Розглянуто методи оптимізації роботи з надлишковим об’єктним кодом за допомогою варіативної мови, що дозволяє знизити складність і підвищити ефективність використання шаблонів. Запропоновано кілька стратегій обробки шаблонів, зокрема простий багатопрохідний і двопрохідний алгоритми, які демонструють різні рівні ефективності залежно від складності завдань. Окреслено структуру згенерованого файлу, яка забезпечує зручність подальшої інтеграції в проєкт, і описано процес отримання шаблону, що включає всі етапи від створення моделі до генерації коду.

Важливим аспектом розділу є демонстрація застосування запропонованого підходу на практиці, що підтверджує його ефективність у вирішенні типових проблем варіативності та оптимізації процесів генерації коду. Отримані результати свідчать про значний потенціал розроблених методів і підходів у покращенні якості та швидкості розробки програмного забезпечення.

ВИСНОВКИ

В магістерській роботі розглянуто моделі, методи та алгоритми покращення ефективності шаблонів

Дослідження спрямоване на вирішення центрального питання: "Чи можливо інтегрувати синтаксичні правила та стандарти кодування в текстові шаблони без створення клонів коду або розсіяного коду?". Для цього було проаналізовано три ключові аспекти:

- Структура та відношення в мовах шаблонів. Встановлено, що деякі структури та відношення, властиві мовам об'єктного коду, важко виразити в мовах шаблонів. Зокрема, це стосується відношень, що призводять до клонування умов (PS1, PS2, PS3) та багаторазової реалізації правил угоди (PS4).

- Моделювання структури та відношень. Запропоновано підхід до моделювання відсутніх відношень за допомогою проміжної мови варіантів (IVL), яка дозволяє явно визначати залежності між фрагментами коду. Це усуває необхідність клонування умов та покращує зручність супроводу шаблонів.

- Розв'язання моделі та генерація коду. Розроблено два алгоритми для обробки IVL та генерації коду, що відповідає синтаксичним правилам та стандартам кодування. Перший алгоритм характеризується простотою, але потенційно великою кількістю проходів по коду. Другий алгоритм є більш ефективним, але вимагає складнішого аналізу даних.

Дослідження показало, що інтеграція синтаксичних правил та стандартів кодування в текстові шаблони можлива шляхом явного моделювання залежностей між фрагментами коду та використання конвеєра модельно-орієнтованої розробки (MDE) для розділення різних аспектів генерації коду. Запропоновані методи дозволяють уникнути клонування коду та розсіяного коду, покращуючи зручність супроводу та повторного використання шаблонів.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
2. Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.
3. Schmidt, D. C., Stal, M., Rohnert, H., & Buschmann, F. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Wiley, 2000.
4. Alexander, C., Ishikawa, S., & Silverstein, M. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, 1977.
5. Freeman, E., Robson, E., Sierra, K., & Bates, B. Head First Design Patterns. O'Reilly Media, 2004.
6. Shalloway, A., & Trott, J. R. Design Patterns Explained: A New Perspective on Object-Oriented Design. Addison-Wesley, 2001.
7. Pree, W. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1994.
8. Vlissides, J. Pattern Hatching: Design Patterns Applied. Addison-Wesley, 1998.
9. Marco Emrich, Ulrich Eisenecker, and Stan Jarzabek. \Generative Programming Using Frame Technology". Master. University of Applied Sciences Kaiserslautern, 2003.
10. Erich Gamma et al. Design patterns: elements of reusable object-oriented software. Pearson Education India, 1996, p. 395. isbn: 020163361-2. doi: 10.1093/carcin/bgs084.
11. Larman, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, 2004.
12. Taylor, R. N., Medvidovic, N., & Dashofy, E. M. Software Architecture: Foundations, Theory, and Practice. Wiley, 2009.

13. Buschmann, F., Henney, K., & Schmidt, D. C. Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing. Wiley, 2007.
14. Gamma, E. Design Patterns in Java. Addison-Wesley, 1998.
15. Johnson, R. E. Frameworks = (Components + Patterns). ACM, 1997.
16. Martin, R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Prentice Hall, 2017.
17. Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003.
18. Beck, K. Implementation Patterns. Addison-Wesley, 2007.
19. Martin, R. C. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2002.
20. Sourour, A. The Design of Modern C++ APIs. Addison-Wesley, 2015.
21. Rising, L., & Manns, M. L. Fearless Change: Patterns for Introducing New Ideas. Addison-Wesley, 2004.
22. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, 1996.
23. Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.
24. Eunsuk Kang and Mark Aagaard. \Improving the Usability of HOL Through Controlled Automation Tactics". In: Theorem Proving in Higher Order Logics. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 157{172. doi: 10.1007/978-3-540-74591-4_13. url: http://link.springer.com/10.1007/978-3-540-74591-4%7B%5C_%7D13.
25. Kuldeep Kumar, Stan Jarzabek, and Daniel Dan. \Managing big clones to ease evolution: Linux kernel example". In: 2016 Federated Conference on Computer Science and Information Systems (FedCSIS) 8 (2016), pp. 1727{1736. doi: 10.15439/2016F173.

26. Hunt, A., & Thomas, D. The Pragmatic Programmer: Your Journey to Mastery. Addison-Wesley, 1999.
27. Alur, D., Crupi, J., & Malks, D. Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall, 2003.
28. Bishop, J. Java Gently: Programming Principles Explained. Addison-Wesley, 2001.
29. Theriault, A. Programming Algorithms in JavaScript. O'Reilly Media, 2017.
30. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. Compilers: Principles, Techniques, and Tools. Pearson, 2006.
31. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms. MIT Press, 2009.
32. Pressman, R. S., & Maxim, B. R. Software Engineering: A Practitioner's Approach. McGraw-Hill, 2014.
33. Sommerville, I. Software Engineering. Addison-Wesley, 2015.
34. Coplien, J. O. Advanced C++: Programming Styles and Idioms. Addison-Wesley, 1992.
35. Erickson, J., & Boehm, B. W. A Framework for Improving Software Development Productivity. Wiley, 2005.
36. Papalambros, P. Y., & Wilde, D. J. Principles of Optimal Design: Modeling and Computation. Cambridge University Press, 2000.
37. Broy, M., & Krüger, I. H. Automotive Software Engineering. Springer, 2005.
38. Rozanski, N., & Woods, E. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley, 2011.
39. Larman, C., & Basili, V. R. Iterative and Incremental Development: A Brief History. IEEE Software, 2003.
40. Kelly, D., & Stachowiak, L. Effective Software Testing. Addison-Wesley, 2002.
41. Rajlich, V. Software Engineering: The Current Practice. CRC Press, 2012.
42. Kamthan, P. Designing Reusable and Modular Software. Pearson, 2008.

43. Simon, H. A. The Sciences of the Artificial. MIT Press, 1996.
44. Love, M. Programming with GNU Software. O'Reilly Media, 1997.
45. Hohpe, G., & Woolf, B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2004.
46. Pollice, G., Augustine, S., Lowe, A., & Madeyski, L. Software Development: Agile Practices for Mission-Critical Projects. Pearson, 2008.
47. Dimitrios Kolovos, Richard Paige, and Fiona Polack. \The epsilon transformation language". In: ICMT 8 (2008), pp. 46{60.
48. Jain, A. Data-Oriented Design in Game Development. CRC Press, 2016.
49. Moo, P. Efficient C++: Performance Programming Techniques. Addison-Wesley, 2005.
50. Lakos, J. Large-Scale C++ Software Design. Addison-Wesley, 1996.