КВАЛІФІКАЦІЙНА РОБОТА

Група МІПЗс-22

Скірчук В.В.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Скірчука Владислава Васильовича

УДК 004.4

Імплементація підходу на основі моделей для побудови гнучких та адаптивних програмних рішень та сервісів

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль

_____Стисло О.В.

(підпис, дата, розшифрування підпису)

Студент

_____ Скірчук В.В.

(підпис, дата, розшифрування підпису)

Допускається до захисту

Завідувач кафедри

_____к.т.н., доц. Ващишак С.П.

(підпис, дата, розшифрування підпису)

Керівник роботи

_____ к.т.н., доц. Демчина М.М.

(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «магістр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

« 19 » лютого 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Скірчуку Владиславу Васильовичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Імплементація підходу на основі моделей для побудови гнучких та адаптивних програмних рішень та сервісів

керівник роботи:

Демчина Микола Миколайович, кандидат технічних наук, доцент

затверджена наказом вищого навчального закладу від « 26 » червня 2023 року

№ 32/1 с

2. Термін подання студентом роботи 16.02.2024

3. Вихідні дані роботи: Формальні моделі, методи та алгоритми.

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

1. Аналіз технологій побудови контекстно-адаптивних програмних рішень.
2. Опис підходу адаптивного моделювання на основі просторі рішень.
3. Дослідження інструментів моделювання архітектур програмних сервісів
4. Імплементація моделей побудови гнучких та адаптивних архітектур

5. Дата видачі завдання 29.06.2023

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Огляд та аналіз технологій побудови контекстно-адаптивних програмних рішень	26.09.2023	Виконано
2.	Опис підходу адаптивного моделювання на основі просторі рішень	20.10.2023	Виконано
3.	Дослідження інструментів моделювання архітектур програмних сервісів	15.11.2023	Виконано
4.	Імплементація моделей побудови гнучких та адаптивних архітектур	30.11.2023	Виконано
5.	Формування висновків	09.12.2023	Виконано
6.	Оформлення пояснювальної записки	22.12.2023	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	11.01.2024	Виконано

Студент _____ Скірчук В.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Демчина М.М.
(підпис) (прізвище та ініціали)

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

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу
16	Історія повторного використання програмного забезпечення	61	Граматика мови правил VecisionKing, яка використовується для визначення залежностей між рішеннями
20	Моделі варіативності як засіб подолання розриву між клієнтами та розробниками	62	Редактор мови правил
23	Спільність і варіативність між різними продуктами	63	Генерація компілятора мови правил і оцінка моделей за допомогою JBoss Rule Execution Engine

АНОТАЦІЯ

Кваліфікаційна робота присвячена виконанню імплементації підходу на основі моделей для побудови гнучких та адаптивних програмних рішень та сервісів шляхом розробки архітектур програмного забезпечення, які підтримують динамічні коаліції програмних служб шляхом введення нових методів складання гетерогенних компонентів і сертифікації властивостей цих композицій.

В першому розділі проаналізовано технології побудови контекстно-адаптивних програмних рішень та сервісів, наведено концепції повторного використання та варіативність програмного забезпечення, досліджено процес варіативності програмного рішення. Наведені сучасні підходи до моделювання програмних систем та сервісів

В другому розділі проведено наведені підходи до моделювання гнучких та контекстно-адаптивних програмних рішень та сервісів. описано підходи адаптивного моделювання на основі просторі рішень, наведені структури моделей рішень предметної області та атрибутів для контекстно-адаптивних програмних рішень.

В третьому розділі здійснена побудова та імплементація моделей побудови гнучких та контекстно-адаптивних архітектур програмних сервісів. виконано дослідження функціональності інструментів моделювання архітектур програмних сервісів, наведено підхід до моделювання гнучких та контекстно-адаптивних архітектур програмного забезпечення і проведена перевірка відповідності та адекватності побудови моделей.

КЛЮЧОВІ СЛОВА: МОДЕЛЮВАННЯ ОЗНАК, ГНУЧКЕ РІШЕННЯ, МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ПРОСТІР РІШЕННЯ, ЕКСПЕРТ ПРЕДМЕТНОЇ ОБЛАСТІ, ВАРІАТИВНІСТЬ, АТРИБУТ, МЕТАМОДЕЛЬ.

SUMMARY

The qualification work is devoted to the implementation of a model-based approach for building flexible and adaptive software solutions and services by developing software architectures that support dynamic coalitions of software services by introducing new methods of assembling heterogeneous components and certifying the properties of these compositions.

In the first section, the technologies of building context-adaptive software solutions and services are analyzed, the concepts of reuse and software variability are given, and the process of software solution variability is investigated. Modern approaches to modeling software systems and services are given

In the second section, the following approaches to modeling flexible and context-adaptive software solutions and services are presented. adaptive modeling approaches based on decision spaces are described, the structures of domain decision models and attributes for context-adaptive software solutions are given.

In the third section, the construction and implementation of models for building flexible and context-adaptive architectures of software services is carried out. the study of the functionality of modeling tools for software service architectures was performed, an approach to modeling flexible and context-adaptive software architectures was given, and a check of the conformity and adequacy of the construction of models was carried out.

KEY WORDS: SIGN MODELING, FLEXIBLE SOLUTION, SOFTWARE MODELING, SOLUTION SPACE, DOMAIN EXPERT, VARIABILITY, ATTRIBUTES, METAMODEL.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

DOPLER - Decision-oriented Product Line Engineering for Effective Reuse

ADL - Architecture Description Language

CBSE - Component Based Software Engineering

CL2 - Caster Level 2 Automation

CTL - Computation Tree Logic

DoVML - Decision-oriented Variability Modeling Language

DSL - Domain Specific Language

EBNF - Extended Backus-Naur Form

FODA - Feature-oriented Domain Analysis

FORM - Feature-oriented Reuse Method

GMF - Eclipse Graphical Modeling Framework

GPL - General Purpose Language

GUI - Graphical User Interface

IoC - Inversion of Control

KLoC - 1000 Lines of Code

MDA - Model-Driven Architecture

MDE - Model Driven Engineering

MDSD - Model-Driven Software Development

MOF - Meta Object Facility

MSDS - Multi-stakeholder Distributed Systems

OCL - Object Constraint Language

OMG - Object Management Group

OSGi - Open Services Gateway initiative

OVM - Orthogonal Variability Modeling

PIM - Platform-Independent Model

PLE - Product Line Engineering

PLUSS - Product Line Use case modeling for System and Software engineering

PSM - Platform-Specific Model

PuLSE - Product Line Software Engineering

RSEB - Reuse-Driven Software Engineering Business

SE - Software Engineering

SME - Situational Method Engineering

SPLE - Software Product Line Engineering

ВСТУП

Актуальність дослідження. Традиційні способи побудови програмного забезпечення, тобто розгляд проектів програмного забезпечення як незалежних один від одного, вже недостатні для вирішення завдань, з якими стикається індустрія програмного забезпечення. Необхідно зв’язати різні продукти організації таким чином, щоб їх спільність (маркетингові, технічні вимоги або вимоги кінцевого користувача) можна було поділити між різними продуктами та проектами. Дослідники та практики погоджуються, що повторне використання програмного забезпечення є ключем до отримання конкурентної переваги, а варіативність є ключем до повторного використання програмного забезпечення.

Оскільки розмір і складність програмних систем збільшуються, дизайн, специфікація та аналіз загальної системної структури стає критичною проблемою. Структурні питання включають організацію системи як композиції компонентів; глобальні структури управління, протоколи зв'язку, синхронізації та доступу до даних; призначення, функціональність елементів дизайну; склад дизайну, елементи; фізичний розподіл; масштабування та продуктивність, і виміри еволюції системи. Це рівень проектування архітектури програмного забезпечення.

За останнє десятиліття архітектурний дизайн виник як важлива підгалузь інженерії програмного забезпечення. Практики прийшли до розуміння того, що хороший архітектурний дизайн є критично важливим фактором успіху для розробки складної системи. Хороша архітектура може допомогти забезпечити систему ресурсами, задовольнить ключові вимоги в таких сферах, як продуктивність, надійність, портативність, масштабованість і сумісність. Практики також почали визнавати цінність прийняття чітких архітектурних рішень і використання минулих архітектурних проектів при розробці нових продуктів. Сьогодні існує багато книг з архітектурного

проектування, регулярні конференції та семінари, присвячені спеціально архітектурі програмного забезпечення, зростає кількість комерційних інструментів для допомоги в аспектах архітектурного проектування.

Кодифікація архітектурних принципів, методів і практик почало призводити до повторюваних процесів архдизайну, побудови критеріїв для досягнення принципових компромісів між архітектурами та стандарти для документування, перегляду та впровадження архітектур.

Якщо ми поглянемо на загальне використання терміну «архітектура» в програмному забезпеченні, ми побачимо, що воно часто використовується різними способами що ускладнює розуміння реальних аспектів застосування. Типове визначення таке: «архітектура програмної або обчислювальної системи - це структури системи, які містять програмні компоненти, зовнішні видимі властивості цих компонентів і зв’язки між ними.

Хоча існує багато схожих визначень архітектури програмного забезпечення, в основі всіх них лежить уявлення про те, що архітектура системи описує її грубу структуру за допомогою одного або кількох переглядів. Структура включає набір дизайнерських рішень верхнього рівня, включаючи представлення того, як система складається з взаємодіючих частин, де знаходяться основні шляхи взаємодії та які ключові елементи :

1. Конструктивна складова: надається архітектурний опис, частковий план розвитку із зазначенням основних програмних компонент та залежності між ними. Наприклад, багаторівневе подання архітектури зазвичай документує межі абстракції між частинами реалізації системи, чітке визначення основних внутрішніх інтерфейсів системи та обмеження,щодо того, які частини системи можуть покладатися на послуги, що надаються іншими частинами.

2. Еволюційна складова: архітектура програмного забезпечення може розкривати розміри, за якими система повинна розвиватися. Зробивши явним «несучі стіни» системи, спеціалісти з її супроводу можуть краще зрозуміти наслідки змін, а отже, і більше точно оцінити вартість модифікацій.

Крім того, архітектурні описи відокремлюють проблеми про функціональність компонента з способами, якими цей компонент пов’язаний (взаємодіє з) іншими компонентами, шляхом чіткого розмежування між компонентами та механізмами, які дозволили їм взаємодіяти. Це розділення дозволяє легше змінювати механізми підключення, вирішувати зростаючі проблеми щодо продуктивності та повторного використання.

Мета і задачі дослідження. **Метою роботи** є дослідження моделей побудови програмних рішень та розробка методів підвищення рівня життєздатної архітектури програмного рішення як ключової віхи в процесі розробки промислового програмного забезпечення, більш чіткого розуміння вимог, стратегій впровадження і потенційних ризиків.

Для досягнення поставленої мети необхідно розв’язати такі задачі:

- виконати аналіз технологій побудови контекстно-адаптивних програмних рішень та сервісів;
- здійснити дослідження процесу варіативності програмного рішення;
- розробити підходи до моделювання гнучких та контекстно-адаптивних програмних рішень та сервісів;
- виконати побудову моделей контекстно-адаптивних архітектур програмних рішень;
- здійснити перевірку відповідності та адекватності побудови моделей.

Об’єктом дослідження є технології побудови гнучких та контекстно-адаптивних програмних рішень та сервісів.

Предметом дослідження є моделі, методи та підходи до моделювання та побудови контекстно-адаптивних програмних рішень та сервісів.

Методи дослідження базуються на використанні методів архітектурних описів, що служать засобом спілкування між зацікавленими сторонами, дозволяють зацікавленим сторонам висловлювати свої думки щодо відносних

ваг характеристик і атрибутів якості коли необхідно враховувати архітектурні компроміси.

Наукова новизна одержаних результатів полягає в розробці моделі контекстно-адаптивної архітектури програмного рішення, яка складається з трьох компонентів: елементів, форми та обґрунтування. Форма визначена в термінах властивостей та зв'язків між елементами – тобто обмежень накладених на елементи. Визначене співвідношення забезпечує базову основу для архітектури в термінах системних обмежень, які найчастіше випливають із системи в формі вимог. Окреслено компоненти моделі в контексті загальної архітектури та виділено архітектурні стилі та представлено розширений приклад для ілюстрації деяких важливих архітектурних елементів.

Практичне значення одержаних результатів полягає в розробці архітектур програмних рішень, які підтримують динамічні коаліції програмних служб шляхом введення нових методів складання гетерогенних компонентів і сертифікації властивостей цих композицій. Пропоновані архітектури програмного забезпечення адаптуються до своїх фізичних налаштувань.

Апробація результатів дослідження. Матеріали дослідження було представлено у матеріалах І Всеукраїнської науково-практичної інтернет конференції "ІТ екосистема: цифровізація бізнес-процесів в умовах війни", у тезах доповіді "Оцінка динаміки модель-базованої розробки".

Структура. Кількість розділів – 3. Загальний обсяг основної частини – 89 сторінок. Список використаних джерел містить – 48 позицій.

РОЗДІЛ 1. АНАЛІЗ ТЕХНОЛОГІЙ ПОБУДОВИ КОНТЕКСТНО-АДАПТИВНИХ ПРОГРАМНИХ РІШЕНЬ ТА СЕРВІСІВ

1.1 Аналіз концепції повторного використання та варіативність програмного забезпечення

Повторне використання програмного забезпечення не є новою концепцією. Як показано на рисунку 1.1 повторне використання програмного забезпечення практикується з перших днів розробки програмного забезпечення. Однак слід зазначити, що рівень абстракції щодо повторного використання постійно зміщується від низькорівневих програмних конструкцій (підпрограм до об’єктів) до компонентів. Підвищення ступеня повторного використання в інженерії програмного забезпечення привертає інтерес як дослідників, так і практиків протягом тривалого часу, і було запропоновано численні методи та інструменти. Серед цих програмних продуктів особливо цікаві. Існує кілька визначень ліній програмних продуктів. Одне із загальноприйнятих визначень таке: « лінійка продуктів — це набір програмно-інтенсивних систем, які мають загальний керований набір функцій, які задовольняють потреби певного сегмента ринку чи місії та які розроблені із загального набору основних програмних атрибутів. у встановлений спосіб».

Розробка лінійки програмних продуктів (PLE) — це дисципліна створення та керування лінійками програмних продуктів. PLE має на меті скоротити вартість і підвищити продуктивність і надійність за рахунок повторного використання артефактів і процесів у певних областях. Було продемонстровано, що PLE є успішним підходом до реалізації потенціалу повторного використання програмного забезпечення. PLE все більше розглядається як ключовий стратегічний підхід до досягнення та підтримки

унікальних конкурентних позицій та задоволення потреб у виході на ринок, вартості, продуктивності та якості в багатьох бізнес-середовищах.

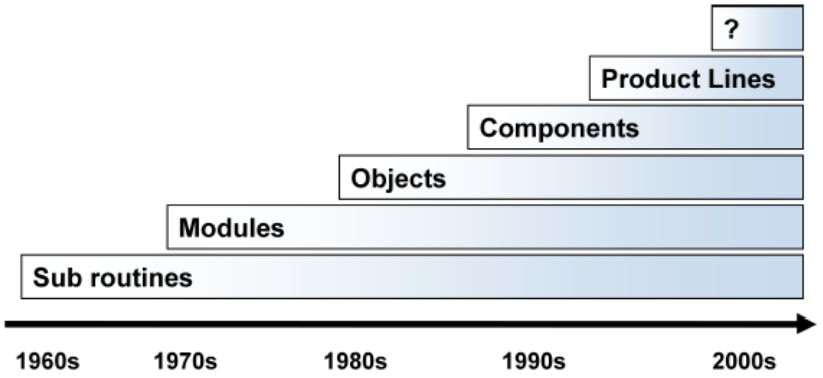


Рисунок 1.1 – Історія повторного використання програмного забезпечення

Сьогодні багато лінійок програмних продуктів розробляються та обслуговуються з використанням модельного підходу. Для моделювання лінійки продуктів доступні численні підходи, наприклад, мови моделювання, орієнтовані на функції, підходи до моделювання змінності, орієнтовані на прийняття рішень, підходи до моделювання варіативності на основі UML, мови моделювання архітектури або ортогональні підходи. На основі цих моделей розроблено численні інструменти для автоматизації діяльності з розробки домену та додатків. Незважаючи на успішність цих підходів і чітку тенденцію (відходу від єдиних систем) до підходів, орієнтованих на змінність, у промисловості все ще існує кілька завад, які перешкоджають широкому впровадженню. Причина цих проблем полягає в негнучкості існуючих підходів до моделювання змінності та інструментів, які часто не підтримують різноманітні потреби різних організацій. Незважаючи на те, що існує міцний консенсус щодо переваг моделювання варіативності, організаціям залишається складно:

- визначити методи та методи, застосовні для їх конкретного контексту;

- адаптувати ці методи та методи для задоволення конкретних потреб;
- інтегрувати їх поточною практикою, інструментами та стандартами.

Різні організації мають різні цілі, очікування та потреби, тому важко визначити мову або процес моделювання варіативності, який можна застосовувати всіма організаціями. Необхідність гнучкого підходу до моделювання змінності стає очевидною при розгляді різних мов, нотацій моделювання або архітектурних стилів, що використовуються різними організаціями: різні зацікавлені сторони мають різні погляди на систему, розмовляють різними мовами (використовують різні нотації), мають знання про різні аспекти системи та мають різні очікування та цілі. Це ускладнює спілкування між ними, що, у свою чергу, може призвести до неправильного тлумачення інформації та непорозумінь. Тому важливо надати чітко визначені концепції для сприяння спільному розумінню варіативності та її впливу.

Незважаючи на всі досягнення в галузі технологій, інструментів створення програмного забезпечення, ефективних мов програмування та прогресу в інших суміжних галузях, однією з основних проблем, яка забороняє індустріалізацію та повну автоматизацію розробки програмного забезпечення, є той факт, що розробка програмного забезпечення базується на неявних знаннях, тобто якість програмного забезпечення зазвичай залежить від креативності, досвіду та індивідуальних ноу-хау розробників і архітекторів. Неявні знання часто є підсвідомими, інтерналізованими і люди можуть або не можуть усвідомлювати, що вони знають і як вони виконують конкретні завдання. На протилежному кінці спектру знаходиться усвідомлене або явне знання — знання, яке індивід явно і свідомо зберігає в розумовому фокусі та може повідомляти іншим. Рішення, прийняті розробниками програмного забезпечення, все більше впливають на якість системи, оскільки програмне забезпечення відіграє важливу роль у багатьох системах. Наприклад, рішення інженерів програмного забезпечення дедалі більше

обмежують рішення інших зацікавлених сторін у системній інженерії, наприклад, ¬інженерів-механіків та електротехніків або торгового персоналу, щоб назвати лише деякі. Таким чином, пошук нових способів накопичення та обміну знаннями про архітектуру технічного програмного забезпечення має великий потенціал для покращення процесу розробки. Це особливо стосується розробки програмного забезпечення, орієнтованого на повторне використання, наприклад, через сильну потребу в спілкуванні між розробником і «повторним використанням» компонентів програмного забезпечення.

Інженерія на основі моделі. Модель — це спрощення системи, побудованої з передбачуваною метою, яка відповідає на запитання замість фактичної системи. Інженерія, керована моделями (MDE), відноситься до ряду підходів до розробки, які використовують моделі як основний засіб комунікації, документації та автоматизації. Обчислювальні парадигми, керовані моделлю, використовують формальні нотації для захоплення бажаних абстракцій систем у моделях. Подібно до основного принципу в об’єктно-орієнтованих мовах, де все є об’єктом, MDE слідує парадигмі все є моделлю. Модель описана чітко визначеною мовою. Добре визначена мова — це мова з чітко визначеною формою (синтаксисом) і значенням (семантикою), яка придатна для автоматизованої інтерпретації комп’ютером. Предметно-орієнтоване моделювання — це використання спеціальних мов для прямої відповіді на питання, що цікавлять, і є більш інтуїтивно зрозумілим для охоплення намірів залучених зацікавлених сторін. Завдяки використанню предметно-спеціальних понять, розробники моделей мають попередньо визначені абстракції, які безпосередньо представляють концепції з області застосування. Як описано в [4], DSL можна розглядати як «спеціалізовані мови», призначені для вирішення проблем у певних областях. DSL, як правило, підтримують абстракції вищого рівня, ніж мови моделювання загального призначення, тому вони вимагають менше зусиль і менше деталі низького рівня для визначення даної системи.

Існує кілька відомих переваг використання DSL над мовою загального призначення (GPL). Синтаксис конкретного DSL зазвичай використовує природні нотації для домену, уникаючи синтаксичних перешкод (такі перешкоди частіше виникають під час використання GPL). Крім того, використання DSL дозволяє краще перевіряти помилки, оскільки аналізатори, що стосуються конкретної проблеми, можна використовувати для пошуку більше помилок, ніж аналогічні аналізатори для GPL. Про помилки можна повідомляти мовою, знайомою експерту домену. Оскільки необхідно враховувати лише вимоги конкретної організації, мові моделювання легше розвиватися у відповідь на зміни в домені. DSL приховують від програмістів деталі мови програмування нижчого рівня, такі як складні структури даних, складні алгоритми та нудний синтаксис GPL.

В даній роботі зроблено спробу скоротити розрив між розробниками, архітекторами, персоналом і клієнтами шляхом накопичення та обміну знаннями про архітектуру та продаж програмного забезпечення в моделях. Отримання та обмін знаннями про архітектуру вже є складним завданням, коли ми маємо справу зі звичайними програмними системами для одного клієнта. У змінному програмному забезпеченні ситуація ще складніша через гнучкість архітектури та складні зв’язки між функціями та компонентами технічного рішення.

Ми досліджуємо, як і де моделі варіативності придатні як спільна база знань, тобто як засіб спілкування між розробниками та клієнтами. Це не тривіально, оскільки люди дивляться на програмне забезпечення з різних точок зору. З цього приводу ми можемо розрізнити два рівні досвіду окремих зацікавлених сторін, залучених до проекту програмного забезпечення – людей, які мають знання про простір рішень і простір проблем. Особи, залучені до простору рішень, мають справу з технічними проблемами, що стосуються впровадження продукту. З іншого боку, особи в проблемному просторі зосереджуються на спробі зрозуміти проблему клієнтів. Здебільшого

це дві різні групи людей - наслідок очевидний: або проблема неправильно зрозуміла, або неправильно проблема вирішена.

Розбіжності між двома групами виникають через різну точку зору, з якої вони розглядають програмне рішення. Інженери-програмісти розуміють реалізацію продукту. Вони розуміють наслідки того, як організовано продукт (його архітектуру чи дизайн), а також типи проблем, які розглядалися під час проектування чи впровадження. Клієнти та продавці, як правило, є користувачами експертних систем. Вони розуміють проблеми клієнтів і чому вирішення цих проблем принесе користь клієнтам. Щоб скоротити розрив (розробник-користувач), ми аналізуємо програмні системи з двох точок зору (рис. 1.2).

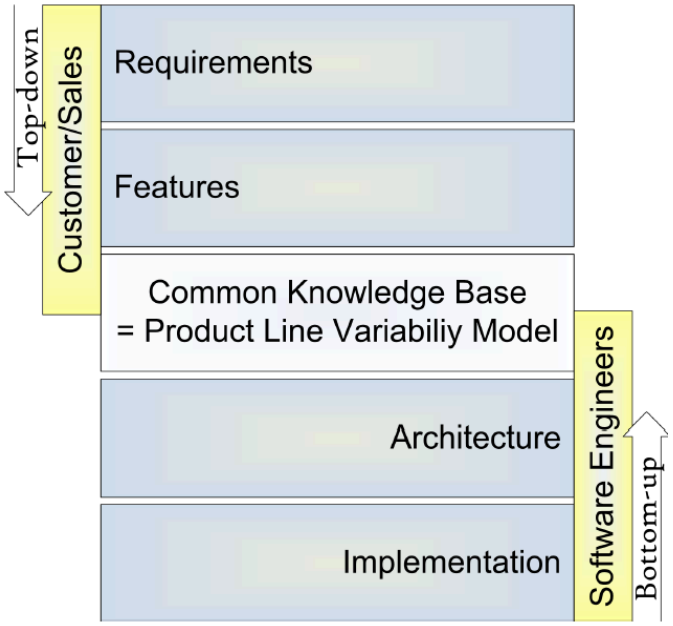


Рисунок 1.2 – Моделі варіативності як засіб подолання розриву між клієнтами та розробниками

Розробники зазвичай розглядають програмне забезпечення з точки зору технічного рішення. Вони сприймають програмне забезпечення як набір компонентів, які вони налаштовують, адаптують і розширюють заздалегідь визначеним способом. Архітектори програмного забезпечення мають більш

абстрактний погляд на конкретне технічне рішення, що лежить в основі, і в основному стурбовані залежностями та співпрацею між компонентами та адаптерами. Відповідальність архітекторів полягає в тому, щоб забезпечити необхідну функціональність і рівень обслуговування, а також реалізувати спеціальні вимоги з мінімальними зусиллями.

Персонал сприймає програмне забезпечення як засіб для задоволення вимог клієнтів. Вони наголошують на відмінності між стандартними функціями, які надає існуючий набір компонентів , і користувальницькими вимогами, реалізованими через часто складні розширення та адаптації. Типовою проблемою є те, що продавці зазвичай не знають про технічні наслідки спеціальних вимог. Команді продажів важко (і навіть необов’язково) зрозуміти всі архітектурні обмеження.

У цій роботі ми фіксуємо неявні знання різних зацікавлених сторін за допомогою моделей. Варіанти рішень клієнтів є основою для підходу зверху вниз. Мета полягає в тому, щоб формалізувати сприйняття клієнтом продукту таким чином, щоб їхні вимоги та побажання можна було формально висвітлити. Для цього важливо розуміти різні рішення, які можуть бути прийняті клієнтами, і залежності між рішеннями. Метою висхідного аналізу є охоплення вже існуючої архітектурної варіативності та отримання технічно здійсненних рішень, які клієнти можуть прийняти в конфігурації продукту. Тому рішення, які має прийняти клієнт, явно моделюються. Модель варіативності, отриману таким чином, може бути об’єднана з властивостями клієнта, таким чином дозволяючи вивести набір варіантів клієнта.

1.2 Дослідження процесу варіативності програмного рішення

Впровадження програмної системи полягає в перетворенні потреб потенційних клієнтів у виконуване програмне рішення. Оскільки враховується все більше і більше вимог клієнтів, система програмного забезпечення стає все більш специфічною для потреб одного конкретного

клієнта. Щоб мати можливість реагувати на мінливі вимоги (або на пізніших етапах розробки, або в процесі еволюції програмного забезпечення), програмні системи все більше проектуються як змінні. Варіативність у програмному забезпеченні описується як «відкладені проектні рішення», де розробник програмного забезпечення дозволяє робити різні варіанти на наступних етапах розробки програмного забезпечення, а не вказувати всі деталі на початку. Однак не всі дизайнерські рішення пов'язані з варіабельністю програмного забезпечення.

Варіативність (мінливість) є результатом різних дизайнерських рішень, які пов’язані з припущеннями/знаннями про різні типи контекстів, які має підтримувати система. Це важлива передумова для успішного застосування підходів на основі повторного використання. Типовий процес розробки програмного забезпечення, орієнтованого на повторне використання, складається з трьох різних дій:

1. Вибір допомагає визначити придатність компонента для використання в передбачуваній кінцевій системі. Він заснований на порівнянні одного компонента з іншими та оцінці його придатності для повторного використання.

2. Адаптація допомагає підготувати компоненти до відповідності їхнім припущенням щодо контексту, у якому вони розгорнуті. Метою адаптації є забезпечення мінімізації конфліктів між компонентами.

3. Збірка — це інтеграція компонентів через чітко визначену інфраструктуру, яка забезпечує зв’язування, що формує систему з окремих компонентів.

Коли система будується шляхом вибору, адаптації та складання існуючих компонентів, можна розрізняти змінні та загальні частини між різними продуктами, створеними з однієї лінії продуктів. Давайте розглянемо набір компонентів, який використовувався для створення трьох різних продуктів. Як показано на рисунку 1.3 , кожен із трьох продуктів має певну окрему функціональність, притаманну продукту, і існують деякі функції, що

збігаються між продуктами. Крім того, продукти також складаються з різних варіацій однієї концептуальної функціональності, що є результатом адаптації використовуваних компонентів. Збірка такої системи без відповідної документації та інструкцій є виснажливим завданням, яке може викликати помилки. Більшість підходів, орієнтованих на повторне використання, здається, мовчки ігнорують той факт, що повторне використання компонента в іншому контексті часто є нетривіальним завданням, особливо коли його потрібно адаптувати відповідно до дещо інших вимог.

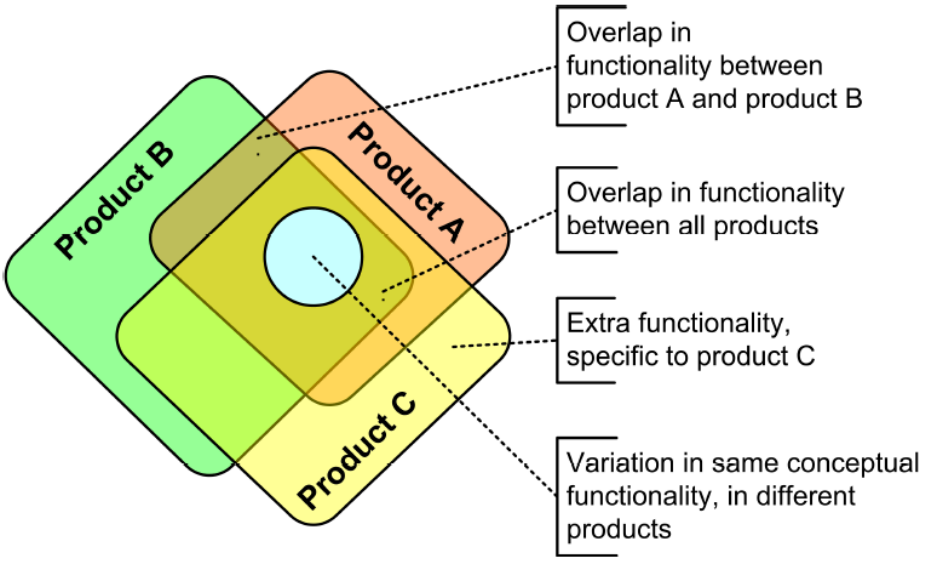


Рисунок 1.3 – Спільність і варіативність між різними продуктами

Налаштувати програмну систему відповідно до особливостей нових доменів і середовищ є складною справою. У багатьох програмних системах (без варіативності дизайну) це технічно неможливо. В інших (де була запланована варіативність) зі збільшенням можливості реконфігурації системи знання, необхідні для управління складністю, також надзвичайно зростають. Це призводить до ситуацій, коли потрібні спеціальні інструменти та методи, щоб допомогти окремим зацікавленим сторонам поділитися своїми знаннями про систему з іншими зацікавленими сторонами.

Робота з варіативністю (впровадження, розуміння та управління змінними системами) не є тривіальною, оскільки залежить від практик розробки програмного забезпечення в конкретному середовищі. Варіативність може бути або спонтанною, або запланованою властивістю систем програмного забезпечення. Це є результатом різних рішень, прийнятих архітекторами та розробниками для задоволення вимог і контекстів різних користувачів. Досвід існуючих великомасштабних систем показує, що знання про мінливість здебільшого неявні природі та проявляється у багатьох різних видах артефактів (документи, компоненти програмного забезпечення, тестові випадки, параметри конфігурації тощо) та різних механізмах, що підтримуються мовами програмування, архітектурними стилями, шаблонами проектування тощо. При плануванні нових систем програмного забезпечення відбивається варіативність у різних варіантах програмної архітектури для виконання вимог замовника та альтернативних механізмах реалізації архітектури.

За даними [5] дискусія про варіативність — це, в основному, спроба знайти відповіді на два запитання щодо повторно використовуваних «атрибутів» або «артефактів».

Що змінюється? Відповідь на це питання також відома як «тема варіативності». Суб’єкт варіативності — це мінливий елемент реального світу або змінна властивість такого елемента, що точно ідентифікує мінливий елемент або властивість реального світу.

Як і де він змінюється? Об’єкт варіативності — це окремий екземпляр суб’єкта варіативності. Об’єкт варіативності використовується для визначення різних «форм» суб’єкта варіативності.

Різні види варіативності мають різний негативний і позитивний вплив. Теоретично добре мати якомога більше варіативності. Основна небезпека полягає в надмірній варіативності яка виходить за рамки, необхідні для позитивного компромісу взаємодії, і яка невиправдано ускладнює процес розробки. Розробники та інженери повинні ретельно розглянути та

обґрунтувати будь-яку дозволену змінність та її вплив на кінцеві продукти. Це можна зробити шляхом явного документування зроблених виборів.

Варіативність важлива для всіх етапів розробки програмного забезпечення, наприклад, варіативність під час проектування дозволяє створити індивідуальне рішення для задоволення потреб окремих клієнтів або груп користувачів. Це звичайна практика в розробці лінійки продуктів, де різноманітність доступних програмних продуктів (основних атрибутів) використовується для створення сімейства продуктів. Варіативність програмних додатків під час виконання є важливою для тривалих програм, таких як веб-сервери або промислові системи автоматизації. Робота зі змінністю під час виконання є важливою, коли зупинка системи для реконфігурації економічно чи технічно неможлива.

Варіативність слід розуміти на різних рівнях (наприклад, вимоги, архітектура або реалізація) і для різноманітних предметно-специфічних артефактів. Важливими аспектами є відстеження між точками варіації, тобто точками прийняття рішень, що описують можливі варіанти щодо функцій або якості атрибутів, а також управління механізмами змінності, які реалізують ці точки. Моделі варіативності охоплюють простір проблеми (потреби зацікавлених сторін і бажані характеристики) і простір її вирішення (архітектура та компоненти технічного рішення). Вони фіксують різні варіанти функцій і компонентів рішення та їх дійсні комбінації, тобто можливі варіанти разом з обмеженнями та залежностями. Моделі змінності також документують фундаментальні загальносистемні рішення щодо конфігурації та створення продукту та обґрунтування цих рішень.

Різні точки зору, з яких мінливість можна класифікувати або проаналізувати називають параметрами варіативності. Різні розміри не обов'язково ортогональні один одному. Існує багато можливих асоціацій, залежностей і взаємозв’язків .

Розрізняють варіативність у просторі (просторова варіативність) і в часі (часова варіативність). Наявність варіативності в просторі означає, що той

самий набір артефактів розробки використовується для отримання кількох програм із різними функціями. Варіативність у часі дається, якщо існують різні версії артефакту, дійсні в різний час. Часовий вимір охоплює зміну змінного артефакту з часом. Вимір простору охоплює одночасне використання змінного артефакту різних форм різними продуктами.

Різні стейкхолдери сприймають різноманітність системи двома способами:

- зацікавлені сторони, орієнтовані на простір проблем, зосереджуються на спробі зрозуміти проблему клієнта,
- стейкхолдери, орієнтовані на простір рішень, мають справу з технічними проблемами, що стоять за тим, як продукт повинен використовуватися. бути реалізованим. Тому роль зацікавленої сторони представляє інший вимір варіативності. Таблиця 1.1 наводить деякі приклади.

Розрізняють внутрішню та зовнішню варіативність програмних систем. Зовнішня — це варіативність артефактів домену, яку бачать клієнти, внутрішня – це варіативність артефактів домену, яка прихована від клієнтів, тобто відома лише розробникам та інженерам.

Таблиця 1.1

Приклад варіативності простору задач і простору рішень

Problem space variability	**Solution space variability**
Support file transfer: Yes | No.	File transfer protocol to be used, e.g., Simple FTP, Secure FTP.
Maximum number of concurrent users to be supported: 100 | 1000 | 10000.	Type of database to be used, e.g., MySQL, Oracle Standard, Oracle Enterprise.
Communication with external services: Yes | No.	Type of communication protocol to be used, e.g., Binary, ASCII, XML.
Support instant messaging: Yes | No.	Chat client to be used, e.g., Skype, ICQ.

Моделювання варіативності. Моделі варіативності були запропоновані як засіб комунікації для роботи з явною документацією неявних знань і кращого використання гнучкості та адаптивності, які забезпечує система. У

багатьох випадках навіть дуже мінлива система не може бути ефективно адаптована до нового середовища (набору вимог) через брак знань про доступну мінливість.

Щоб подолати такі ситуації, неминучим є обмін знаннями між членами команди. Через великий розмір програмних систем сьогодні неможливо, щоб окремі зацікавлені сторони виступали «носіями знань», оскільки кількість таких носіїв обмежена.

Процес документування та визначення варіативності системи, щоб неявні знання в головах різних зацікавлених сторін стали доступними, відомий як моделювання варіативності. Обсяг і вид інформації, яка міститься в моделі варіативності, залежить від мотивів, що стоять за її мотивами моделювання варіативності. Як, наприклад, метод аналізу домену, орієнтованого на особливості, підтримує ідентифікацію помітних або відмітних видимих користувачем функцій у класі пов'язаних ¬систем програмного забезпечення. Інші підходи, такі як [7] і поетапна конфігурація моделей ознак [8], зосереджені на використанні моделей варіативності для цілей конфігурації.

Ретельний аналіз сучасного стану та семінари з експертами з галузі показали, що в основному необхідно враховувати 6 різних аспектів варіативності (рис. 1.4).

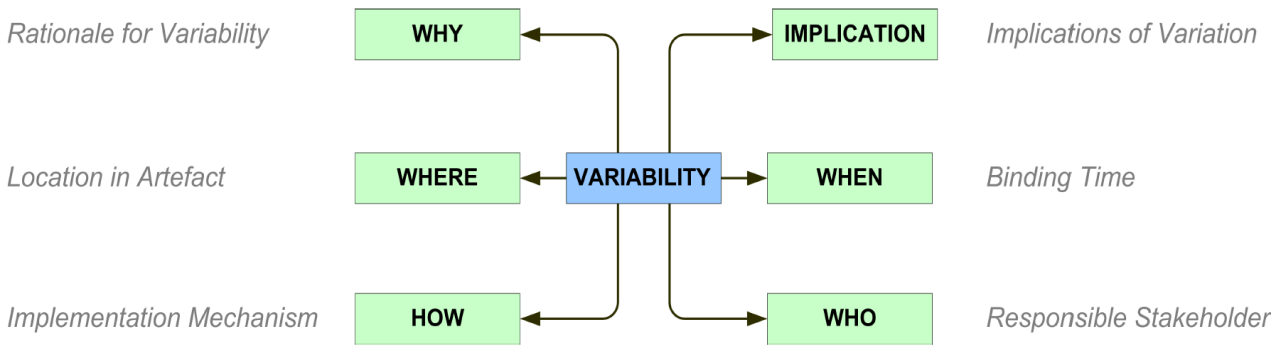


Рисунок 1.4 – Різні аспекти варіативності, які повинні бути охоплені моделлю

ЧОМУ. Обґрунтування варіативності є, мабуть, найважливішим аспектом моделювання, оскільки отримання такого роду знань навряд чи можна автоматизувати. Наприклад, виробник мобільного телефону може вирішити, чи встановити в певну модель протокол зв'язку Bluetooth або WI-FI модуль. Обґрунтуванням цього може бути необхідність звернення до різних груп користувачів із різними робочими середовищами. Модель варіативності повинна відповідати на запитання: «чому потрібна певна варіація?».

ДЕ. Крім того, чому система була розроблена як змінна, важливо знати, які артефакти розвитку пов’язані з мінливістю. Залежно від того, як реалізовано змінність , може бути декілька артефактів, які реалізують одну конкретну проблему змінності. Типи цих артефактів відрізняються в різних областях. Точка в просторі артефакту, де виникає змінність, називається точкою варіації.

ЯК. Механізми, які використовуються для реалізації необхідної варіативності, пов’язані з різними видами предметно-специфічних артефактів. Ці механізми варіюються від конструкцій низького рівня, таких як параметри конфігурації, методи рефлексивного програмування або умовної компіляції, до методів більш високого рівня проектування, таких як фреймворки плагінів, мета-інструменти та фреймворки додатків.

ЗНАЧЕННЯ. Наслідки впровадження варіативності можна розглядати як виконання обґрунтування цієї конкретної точки варіації. Однак це не завжди зрозуміло, оскільки наслідки прийняття рішень щодо змінності (наприклад, зміна параметрів конфігурації або адаптація умовного шаблону компіляції) явно не задокументовано. Таким чином, модель варіативності також повинна відповідати на запитання, які наслідки прийняття рішень, пов’язаних із мінливістю системи.

КОЛИ. Визначаючи етап у процесі розробки, можна вказати час, який підходить для прийняття рішень щодо конфігурації. Наприклад, рішення, пов’язані з високою функціональністю системи (наприклад, чи буде USB-програвач вбудований в автомобіль) можуть бути прийняті під час

розробки, деякі інші рішення (наприклад, який мережевий протокол використовується для зв’язку) є актуальними під час впровадження, а інші можна використовувати лише під час виконання тощо. Таким чином, документація про час зв’язування для варіативності також є невід’ємною частиною моделі варіативності.

ХТО. Недостатньо знати, чому щось можна змінити і які наслідки змін, якщо ніхто не робить змін. Тому не менш важливо знати, хто (наприклад, на основі ролей зацікавлених сторін, таких як архітектор програмного забезпечення, розробник або продавець) може, може або повинен приймати рішення щодо вибору з доступних варіантів. Іноді такі рішення також можуть автоматично прийматися системою (від імені користувача на основі інших рішень).

1.3 Сучасні підходи до моделювання програмних систем та сервісів

Моделювання ознак є найвидатнішим підходом для моделювання варіативності. Дані щодо проектування лінійки продуктів також показує, що це найбільш інтенсивно досліджений метод моделювання варіативності. Починаючи з FODA (Feature Oriented Domain Analysis), функціонально-орієнтоване уявлення про продуктові лінії вже вийшло далеко за межі моделювання змінності та системної документації. Сьогодні доступні численні варіанти інструментів і методів моделювання змінності на основі ознак, інтерпретації моделей ознак та ін. Загалом, функціональна модель фіксує видимі для зацікавлених сторін характеристики та аспекти лінійки продуктів, такі як функціональні особливості окремих продуктів, атрибути якості програмного забезпечення як лінійки продуктів, так і окремих продуктів, щоб забезпечити огляд можливостей системи.

Розглянемо метод виявлення та представлення спільності між пов’язаними програмними системами. Охоплюючи знання експертів, метод FODA намагається кодифікувати «процеси мислення» які використовуються

для розробки програмних систем у пов’язаному класі чи домені. FODA спочатку розроблявся не для моделювання варіативності лінійки продуктів, а для цілей аналізу домену. Аналіз домену підтримує повторне використання програмного забезпечення, фіксуючи досвід домену, який використовується для підтримки зв’язку, навчання, розробки інструментів, специфікації та дизайну програмного забезпечення.

FODA розглядає функції як атрибути системи, які безпосередньо впливають на кінцевих користувачів. Кінцеві користувачі мають приймати рішення щодо доступності функцій у системі, і вони мають розуміти значення функцій, щоб використовувати систему. Документація функціональної моделі включає структурну діаграму, що показує ієрархічну декомпозицію ознак із зазначенням необов’язкових і альтернативних функцій, визначення ознак і правила їх композиції. Щоб визначити відношення між ознаками, FODA використовує структурний зв’язок який представляє логічне групування ознак. Альтернативні або необов’язкові характеристики кожної групи позначаються в моделі ознак маленькими дугами та колами відповідно. Альтернативні ознаки вважаються спеціалізаціями ознаки більш загальної категорії.

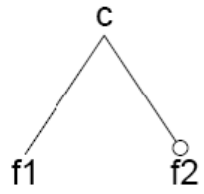
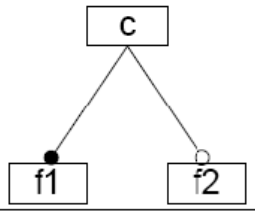
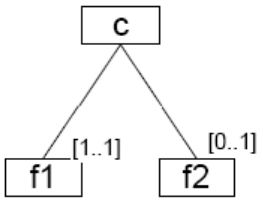
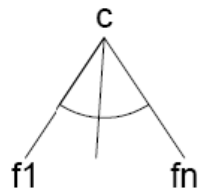
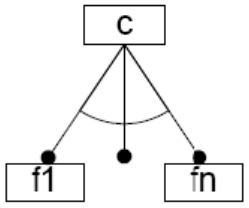
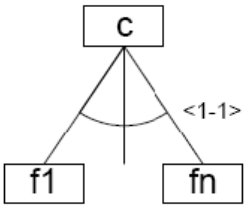
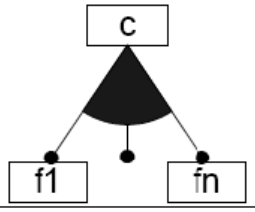
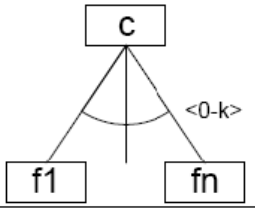
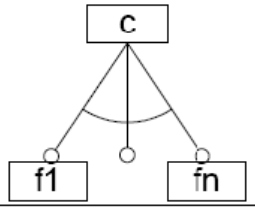
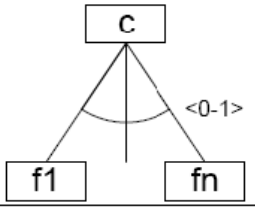
Моделювання ознак на основі кардинальності включає ряд розширень до оригінальної нотації FODA (табл. 1.2). Обмеження на ознаки виражаються за допомогою потужностей, як у моделюванні сутності-зв’язку або моделюванні UML. Модель ознак, заснована на кардинальності – це ієрархія ознак, де кожна ознака має кардинальність ознаки. Автори стверджують, що використання кардинальностей є засобом вираження додаткових обмежень для моделювання функцій і може допомогти, забезпечуючи засоби задоволення обмежень для моделювання функцій та інструменти налаштування на основі функцій.

У моделюванні функцій на основі кардинальності діаграма функцій транслюється на діаграму класів UML. На діаграмі класів сутності відповідають ознакам, і кратність на кінці кожної композиції дорівнює 1, тоді

як кратність на іншому кінці дорівнює відповідній потужності на діаграмі ознак. Формалізм моделювання ознак на основі кардинальності допускає щонайбільше один атрибут на функцію, а тип атрибута є або примітивним типом, або посиланням на іншу функцію.

Таблиця 1.2

Графічне представлення різних нотацій, що використовуються в моделюванні ознак

Original FODA notation	**Czarnecki-Eisenecker notation**	**Extended Czarnecki-Eisenecker notation**
mandatory and optional subfeature 	mandatory and optional subfeature 	mandatory and optional subfeature 
alternative subfeatures 	XOR group 	group with cardinality <1-1> 
	OR group 	group with cardinality <0-k> 
	XOR group with optional subfeatures 	group with cardinality <0-k> 

В [9] описують використання мови обмежень об’єктів (OCL) у контексті моделей ознак для вираження обмежень, які не можна виразити діаграмою. Автори стверджують, що OCL є достатнім для вираження таких обмежень і підтверджують своє твердження низкою зразків і тверджень. Вони також ідентифікують набір можливостей на основі задоволення обмежень, які можуть бути забезпечені моделюванням функцій і інструментами налаштування на основі функцій. Автори також мають на меті верифікацію шаблонів моделей на основі функцій OCL.

Метод FeatuRSEB [10] є комбінацією методу FODA та RSEB (бізнес програмної інженерії керований повторним використанням). RSEB базується на OO Software Engineering [11] та OO Business Engineering [12] які застосовуються до організації, що займається створенням наборів пов’язаних додатків із наборів багаторазових компонентів. FeatuRSEB використовує розширення UML для моделювання та специфікації систем додатків, багаторазових систем компонентів і багаторівневих архітектур, а також для вираження варіативності системи в термінах точок варіації та приєднаних варіантів.

Пізніше в [13] розширено FeatuRSEB для роботи з часом зв’язування, вказуючи, коли функції можна вибрати і зовнішніми функціями, які є технічними можливостями, пропонованими цільовою платформою системи. Підхід PLUSS [14] базується на FeatuRSEB і поєднує діаграми функцій і діаграми варіантів використання, щоб відобразити високорівневе уявлення про сімейство продуктів.

В методі FORM (метод повторного використання, орієнтований на особливості) [15] основна увага приділяється систематичному відкриттю та використанню спільності у відповідних системах програмного забезпечення. FORM розширює FODA на етап проектування програмного забезпечення та визначає, як моделі функцій можна використовувати для розробки архітектури домену та компонентів для повторного використання. Теорія, що лежить в основі цього розширення, полягає в тому, що особливості домену

характеризують кожен варіант продукту в домені, а код, який реалізує характерні функції, повинен бути упакований, керований і повторно використаний як програмні модулі.

Деякі дослідники також представили ідею шаблонів функцій. Шаблон функції складається з моделі функції та анотованої моделі, вираженої на деякій загальній мові моделювання, такій як UML або предметно-орієнтованій мові моделювання [16]. На основі конкретної конфігурації функцій процесор шаблонів створює екземпляр шаблону, оцінюючи умови присутності в моделі та видаляючи елементи, умови присутності яких оцінюються як false. Ключова перевага шаблонів моделей полягає в тому, що вони дозволяють нам підтримувати кілька варіантів моделей, наприклад варіанти бізнес-моделей або моделей дизайну для різних членів лінійки продуктів, у накладеній формі в межах одного артефакту [17].

В дослідженні [18] представляють процес моделювання рішень на основі XML, де модель прийняття рішень представляє всі можливі вимоги користувача, визначені під час аналізу домену, і набір правил і обмежень, пов’язаних з ними. Так само автори визначають модель рішень як «документ, що визначає рішення, які повинні бути прийняті для визначення члена домену». Кожне рішення, представлене в моделі прийняття рішень, визначається набором, що містить таку інформацію:

1. Ім’я, що ідентифікує рішення за допомогою унікального ідентифікатора.
2. Опис, який містить необхідну інформацію для підтримки прийняття рішення.
3. Тип, що виражає можливі значення, підтримувані рішенням.
4. Значення за замовчуванням, яке вказує на значення, яке автоматично впливає на рішення, якщо рішення не прийнято явно.
5. Обґрунтованість, що вказує на критерії прийняття чи неприйняття рішення.

6. Залежності, які явно дозволяють уточнювати зв’язки між різними рішеннями.

Автори чітко розрізняють обмежені та необмежені рішення. Необмежені рішення не підтримують обмежень, окрім обмежень типу даних. Рішення з обмеженнями мають інші специфікації обмежень, що робить їхню специфікацію та реалізацію в документах XML більш складними. Однією з унікальних особливостей цього підходу (порівняно з нашим підходом) є концепція колекції рішень, які є екземплярами рішення або набору рішень. Наприклад, коли потрібні два екземпляри певного компонента, процес прийняття рішень щодо налаштування цих компонентів потрібно повторити для кожного необхідного компонента, тобто рішення потрібно дублювати. Кількість екземплярів можна вказати як обмеження колекції. Підтримуються два типи зв’язків між рішеннями:

- значення рішення може впливати на діапазон або значення іншого рішення;
- значення рішення може впливати на те, чи потрібно приймати інше рішення.

Варіативність на рівні архітектури. Мови опису архітектури (ADL) — це формальні позначення для опису програмних систем. ADL знаходяться на концептуальному використанні мов вимог, програмування та моделювання, але вони відрізняються від усіх трьох. На відміну від мов моделювання вимог, які моделюють простір проблем, ADL моделюють простір рішень. Мови моделювання загального призначення зазвичай зосереджуються на моделюванні внутрішньої структури компонентів, тоді як ADL зазвичай описують взаємодію компонентів. Порівняно з мовами програмування, де машинний код генерується для конкретної платформи чи технології, ADL розроблено таким чином, щоб бути незалежним від технологічних платформ.

Було запропоновано ряд ADL для моделювання архітектур, як у межах конкретної програми, так і як мови моделювання архітектури загального призначення , наприклад, Darwin, xADL [18] тощо. Існує так багато мов, що

не завжди зрозуміло, яка з кількох можливих ADL найкраще підходить для конкретної проблеми. Систематичне дослідження [19] мов опису архітектури показує, що більшість ADL поділяють набір фундаментальних конструкцій і концепцій моделювання, включаючи ¬компоненти, з'єднувачі, інтерфейси та архітектурні конфігурації.

У цій роботі ми розглядаємо ADL внаслідок можливостей моделювання змінності та розширюваності. Більшість доступних сьогодні ADL є монолітними. Їх набори функцій і граматика фіксовані, і додавання нових конструкцій до монолітного ADL неможливо без модифікації набору інструментів, що підтримує ADL.

xADL 2.0 — це мова опису архітектури програмного забезпечення (ADL) для моделювання архітектури програмних систем. На відміну від багатьох інших ADL, xADL 2.0 визначається як набір схем XML. Це надає xADL 2.0 безпрецедентну розширюваність і гнучкість, а також базову підтримку багатьох доступних комерційних інструментів XML. Сама мова xADL 2.0 не прив'язана до якогось певного архітектурного стилю, набору інструментів чи методології. Бібліотеку зв'язування даних можна використовувати окремо, незалежно від будь-якого конкретного середовища розробки чи домену. Поточний набір схем xADL включає підтримку моделювання для:

- елементів системи під час виконання та проектування;
- підтримки архітектурних типів;
- відмінностей архітектури;
- архітектур лінійки продуктів.

XADL визначає основні архітектурні елементи, такі як компоненти, з'єднувачі, посилання або інтерфейси для моделювання архітектури. За допомогою розширень xADL підтримує моделювання компонентів або з'єднувачів архітектури як опціональних або варіантних. Параметри xADL вказують на точки варіації в архітектурі, де структура може змінюватися включенням або виключенням елемента або групи елементів. Варіанти

вказують на точки в архітектурі, де одна з кількох альтернатив може бути замінена елементом або групою елементів. Зв'язки та взаємозалежності між різними архітектурними елементами моделюються за допомогою сутностей які є логічними виразами. Ці сутності складаються зі змінних, значення яких визначають, чи буде певний додатковий компонент включено в остаточну систему чи ні, на основі рішень у конфігурації продукту.

На рисунку 1.5. представлено невеликий приклад підсистеми, яку змодельовано за допомогою xADL.

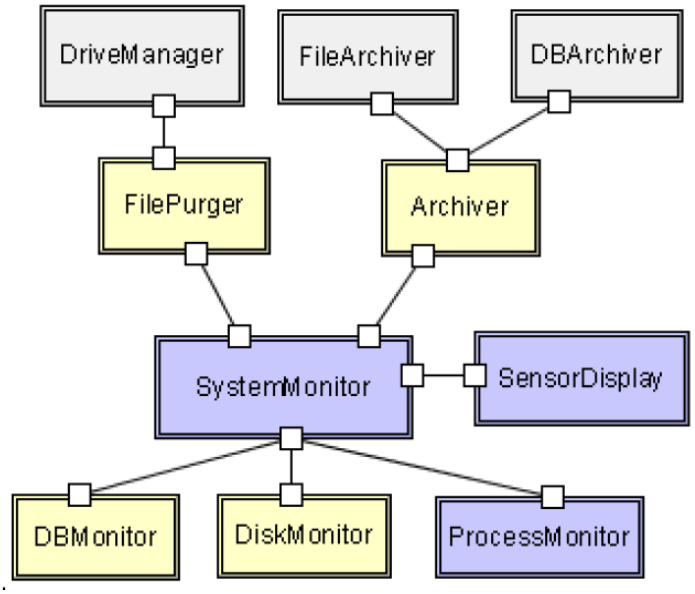


Рисунок 1.5 – Приклад моделі архітектури xADL

Таблиця 1.3

Приклади деяких типових обмежень для моделювання варіативності архітектури моделі xADL, зображеної на рисунку 1.5.

#	**Description**
C1	FilePurger, Archiver, DBMonitor and DiskMonitor are optional.
C2	You can only choose one type of archiver.
C3	FilePurger can be chosen only if DiskMonitor exists.
C4	Without DBArchiver, DBMonitor makes no sense.
C5	Without FileArchiver, DiskMonitor makes no sense.
C6	If DBArchiver is chosen, you must also choose DBMonitor.
C7	If FileArchiver is chosen, DiskMonitor is also needed.

Моделі xADL описують архітектуру з точки зору підархітектур, компонентів, з’єднувачів, інтерфейсів і посилань. xADL підтримує моделювання варіативності за допомогою параметрів, варіантів і версій. Проаналізувавши технічне рішення, визначено необов’язкові та варіативні архітектурні елементи. Зв’язки між змодельованими компонентами та з’єднувачами наведено в таблиці 1.3.

Висновки до розділу 1

В даному розділі проведено аналіз концепції варіабельності програмного забезпечення та сервісів, надано приклади варіативності та визначено різні області застосування для моделей варіативності. Проведено огляд сучасного стану моделювання змінності та аналіз сильних і слабких сторін процесів та концепцій моделювання програмних рішень та сервісів.

РОЗДІЛ 2. ПІДХОДИ ДО МОДЕЛЮВАННЯ ГНУЧКИХ ТА КОНТЕКСТНО-АДАПТИВНИХ ПРОГРАМНИХ РІШЕНЬ ТА СЕРВІСІВ

2.1 Опис підходу адаптивного моделювання на основі просторі рішень

Розглянемо підхід який базується на припущенні, що незважаючи на багато різних практик реалізації в різних областях, основна проблема при описі варіативності включає моделювання проблемного простору (тобто потреб зацікавлених сторін), простору рішень (тобто архітектури та компоненти технічного рішення) і простежуваність між ними.

Моделювання простору рішень вимагає можливостей для фіксації варіативності різноманітних багаторазово використовуваних атрибутів, таких як архітектура, код, тестові випадки, процеси, документи та моделі. Управління варіаціями на різних рівнях абстракції та між усіма загальними артефактами розробки є складним завданням, особливо коли системи, що підтримують різні продукти, дуже великі, як це зазвичай буває в промислових умовах [21]. Мова для моделювання простору рішень має бути гнучкою та адаптованою до практики реалізації в різних областях. Він має бути незалежним від практик реалізації конкретного домену.

Використаємо мову моделювання змінності, орієнтовану на прийняття рішень (DoVML), яка підтримує моделювання простору проблем за допомогою рішень і простору рішень за допомогою атрибутів. На рисунку 2.1 зображено метамодель високого рівня мови моделювання, де ключовими елементами моделювання є рішення та активи. Рішення та активи пов’язані між собою умовами включення.

Основна мета-модель, зображена на рисунку 2.1 є загальною і адаптованою для вирішення предметно-специфічних концепцій. Вона

представлена у формі нових типів разом із їхніми атрибутами та зв'язками між ними (рис. 2.2).

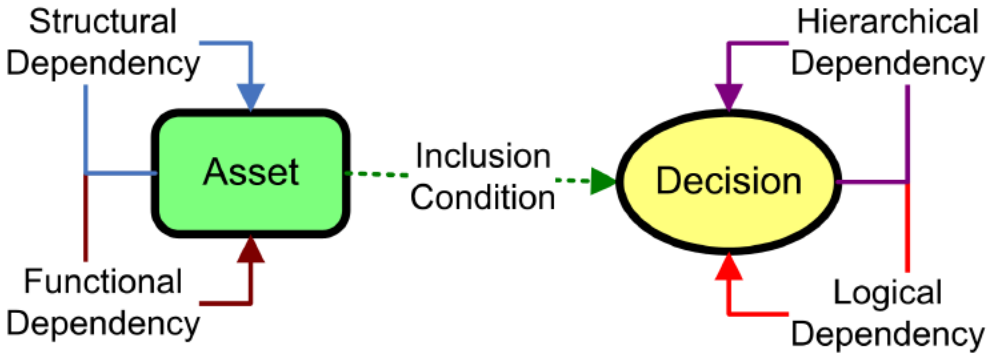


Рисунок 2.1 – Основна мета-модель із зображенням ключових елементів моделювання

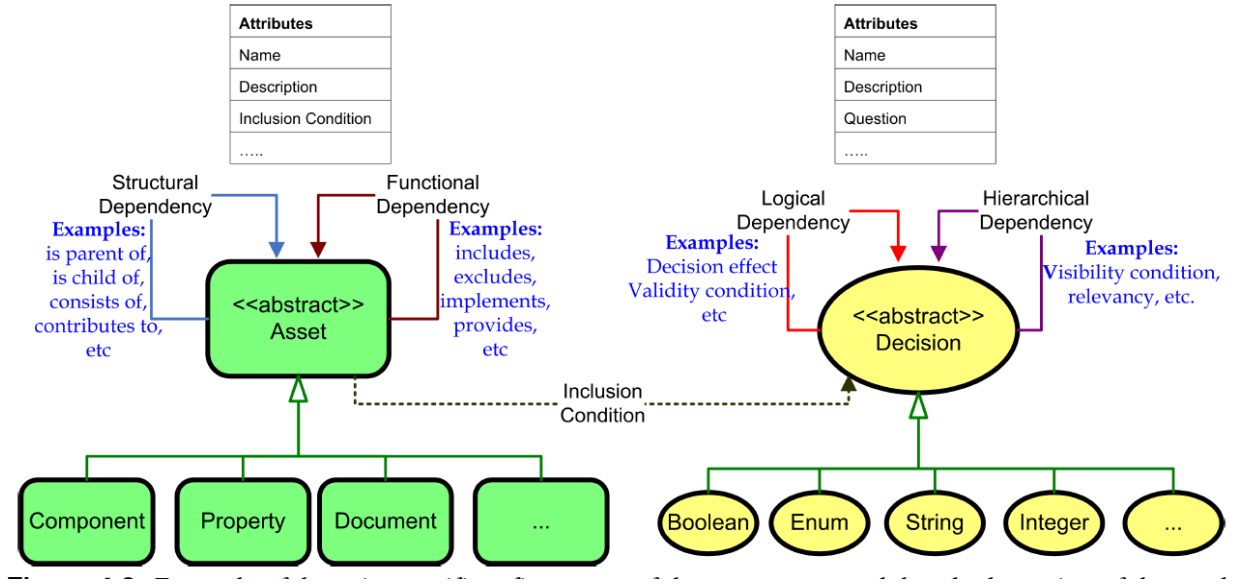


Рисунок 2.2 – Приклад предметних уточнень основної метамоделі та адаптації мови моделювання

Така конфігурованість мови дозволяє адаптувати підхід до потреб різних практик реалізації варіативності. На рисунку 2.3 зображено просту модель варіативності, що складається з рішень та атрибутів як ключових елементів моделювання.

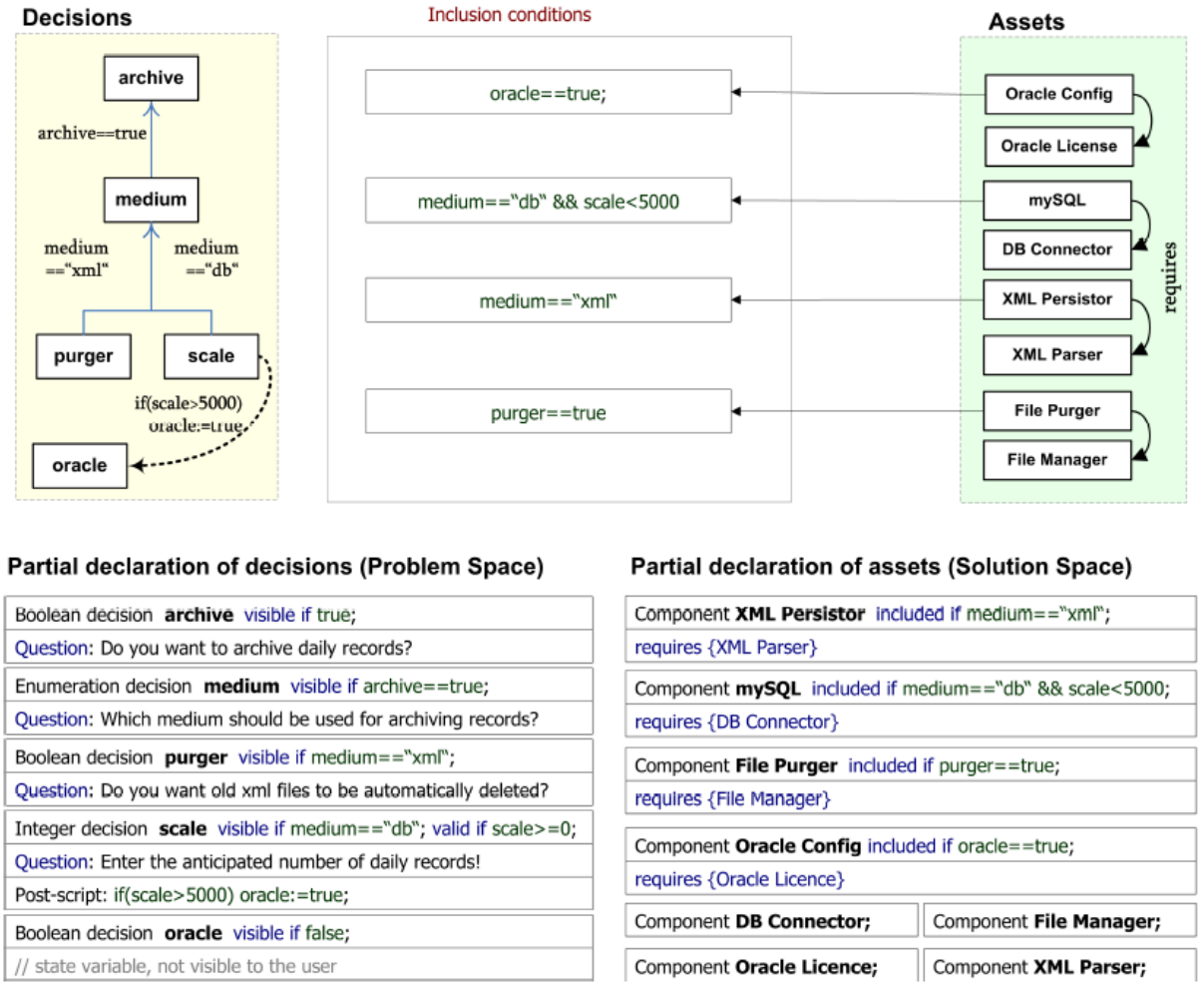


Рисунок 2.3 – Приклад моделі варіативності

Прийняття рішення – це процес оцінки переваг кількох варіантів і вибору одного з них для використання. Результат процесу прийняття рішення призводить до вибору курсу дій серед кількох доступних альтернатив. Варіативність також стосується альтернатив і вибору. Наприклад, вибір певного продукту з лінійки продуктів здійснюється шляхом прийняття набору конфігураційних рішень.

Рішення можна використовувати для представлення точок варіації в моделі лінійки продуктів. Процес прийняття рішення передбачає оцінку переваг кількох варіантів і вибір одного з них для дії (наприклад, на основі врахування вимог замовника). Іншими словами, прийняття рішень призводить до вибору курсу дій серед кількох доступних альтернатив. Рішення не є незалежними одне від одного і не можуть прийматися окремо.

Багато рішень обмежені залежно від контексту вже прийнятих рішень. У нашому підході до моделювання враховуються два типи залежностей між рішеннями: по-перше, не всі рішення є однаково важливими або актуальними в певний час, тому нам потрібні конструкції для моделювання ієрархії рішень. По-друге, прийняття певного рішення може мати наслідки для інших рішень, які також необхідно враховувати. Тому потрібно визначити фактори, які впливають на сам процес прийняття рішень.

Основна мета-модель (рис 2.1) показує ієрархічні залежності, що визначають, як організовані рішення і логічні залежності, що визначають зв’язок між значеннями рішення.

Ієрархічні залежності використовуються для визначення того, коли певне рішення є очевидним для користувача. Ієрархічний порядок рішень додає контекст рішенням, наприклад, не було б сенсу запитувати користувача про потужність системи баз даних, якщо він не має наміру використовувати базу даних.

Логічні залежності - це дії, які необхідно виконати після прийняття рішення. Як правило, це правила, які необхідно перевірити до та після прийняття рішення. Наприклад, тип бази даних, яка буде використовуватися, може бути логічно індукований розміром системи.

2.2 Структура моделей рішень предметної області

Опишемо структуру моделей рішень та значення різних конструкцій неформально без використання формальної семантики. Для простоти прикладів ми припускаємо, що синтаксис моделей рішень подібний до мов програмування Pascal.

Кожне рішення відповідає змінній рішення (порівняній з типізованою змінною в мовах програмування), значення якої встановлюється прийняттям рішень. Імена не мають формального значення, але вони мають величезне практичне значення для читабельності моделі рішень (як і використання

мнемонічних імен у традиційному програмуванні). Рішення специфікуються в моделі рішень шляхом використання наступних деталей (рис 2.4) :

- набір можливих значень (визначених типом рішення) та обмежень значення (визначених умовами дійсності);
- специфікація позиції рішення в ієрархії рішень по відношенню до інших доступних рішень (визначається умовою видимості);
- специфікація наслідків прийняття рішення або впливу на інші рішення (визначається ефектами рішення);
- мітки та анотації, що надають інформацію для користувача, щоб краще зрозуміти рішення (визначається атрибутами рішення).

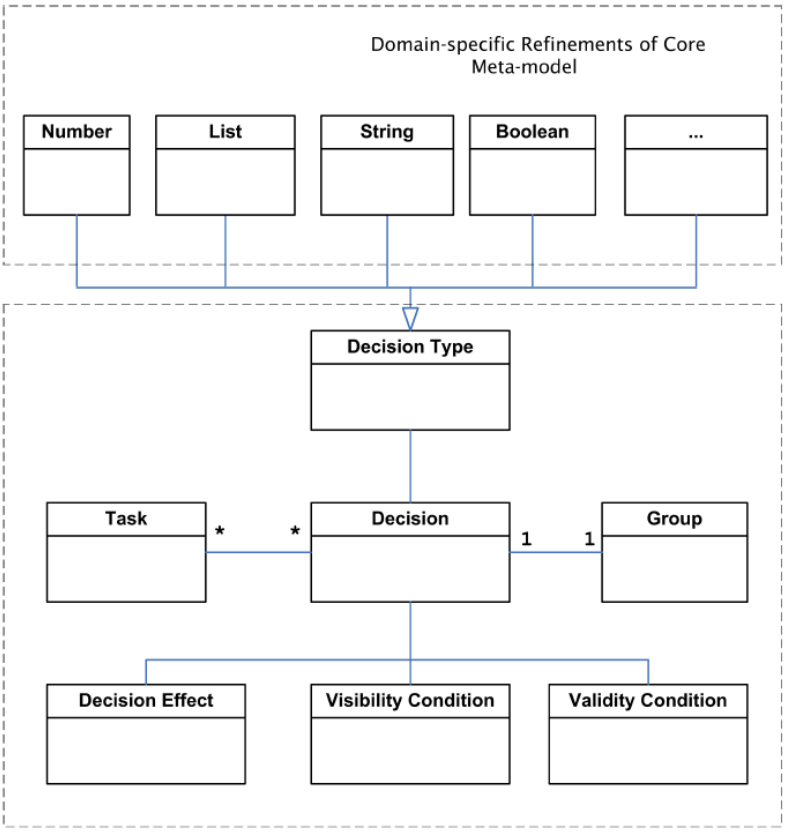


Рисунок 2.4 – Мета-модель для процесу прийняття рішень

Щоб пояснити значення цих конструкцій, використаємо просту модель прийняття рішень, зображену на рисунку 2.5, де зображено чотири рішення та залежності між ними.

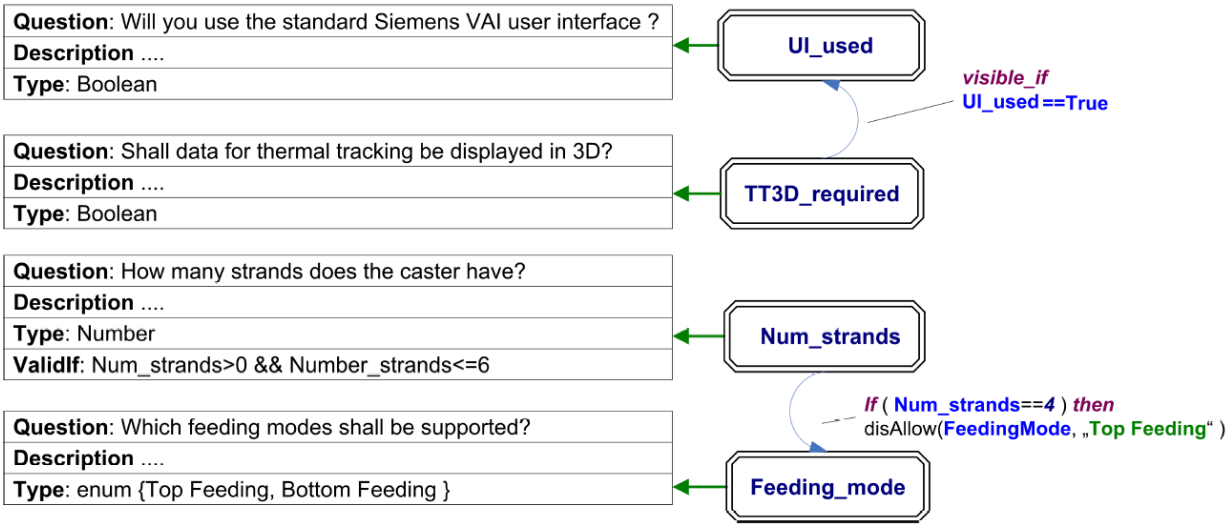


Рисунок 2.5 – Приклад моделі прийняття рішень, що демонструє різні конструкції моделювання

Рішення можна порівняти зі змінною даного типу рішення. Тип рішення обмежує діапазон значень, які можуть бути призначені рішенню. У нашій мові моделювання є три попередньо визначені типи рішень, які представляють логічні значення, рядки та числа. Окрім попередньо визначених типів, користувач може визначати власні типи перерахування, які потрібні для кожної моделі. Ось кілька прикладів:

```
Boolean dbRequired ;
```

Це позначає рішення, де користувач має можливість вибрати True або False для рішення dbRequired. Якщо є рішення щодо кольору продукту, то його тип можна визначити як перелік із зазначенням усіх доступних кольорів. Це означає, що користувач може вибирати між чотирма заданими кольорами та призначати підмножину даного набору як значення кольору рішення:

```
ColorType = { red , blue , black , orange } ;
ColorType color ;
```

На рисунку 2.5 можемо побачити три типи рішень:

```
Boolean UI_used , TT3D_required ;
Number Num_strands ;
FeedType = { Top Feeding , Bottom Feeding } ;
FeedType FeedingMode ;
```

Атрибути рішення – це анотації до рішень, які надають детальну інформацію про рішення. Анотації не мають формального значення, але допомагають зрозуміти модель. Прикладами таких міток є описи, зображення та URL-адреси, запитання, яке задають користувачеві тощо. Використання функцій міток (у порівнянні з неструктурованими текстовими тегами) допомагає краще інтерпретувати теги.

Набір можливих значень рішення, визначений типом рішення, є досить обширним. Як приклад розглянемо числове рішення Speed_min . Відповідно до визначення типу Speed_min усі дійсні числа R є можливими значеннями. Щоб допомогти розробнику моделей обмежити значення рішення, для Speed_min можна вказати умову дійсності .

Умову дійсності рішення можна розглядати як постумову, яка повинна бути виконана після прийняття рішення, що означає, що ця умова стверджується до того, як значення буде присвоєно змінній рішення. Ось кілька прикладів:

- Щоб обмежити значення Speed_min лише додатними числами, можна визначити умову дійсності наступним чином, що означає, що користувач може призначати лише додатні раціональні числа:

```
Speed_min. :
Speed_min.validIf (Speed_min > 0);
```

- Щоб позначити, що користувач повинен вибрати червоний колір, приймаючи рішення щодо кольору, можна вказати:

```
color.validIf (color==ColorType.red) ;
```

- Використовуючи умови дійсності, також можна вказати кілька діапазонів для рішень. Наприклад:

```
decision1.validIf((decision1 >= n1 && decision1 <= n2) ||
(decision1 >= n3 && decision1 <= n4 ) )
```

- На рисунку 2.5, бачимо, що для рішення Num_Strands визначено умову дійсності:

```
Num_Strands.validIf (Num_Strands > 0 && Num_Strands <= 6);
```

Рішення, які спочатку не є видимими для користувача, називаються рішеннями станів і можуть бути пов’язані зі своїми значеннями лише в результаті правил похідних (рішення-наслідки інших рішень). Такі правила допомагають агрегувати значення рішень, які вже були прийняті і дозволяють спростити складні вирази в моделях. Рішення про стан можна використовувати для відстеження різних станів виконання моделі. Наведемо приклад:

- Рішення, яке визначає, чи потрібна база даних Oracle для кінцевої системи, може бути прив’язане до певного значення автоматично після того, як користувач визначиться з розміром кінцевої системи. Користувачеві ніколи не задають питання, чи хоче він базу даних Oracle:

```
Boolean oracle_needed ;
oracle_needed.visibleIf(false);
int size;
size.visibleIf(true ) ;
size.effect{
     if(size>5000) then oracle_needed:=t rue ;
}
```

Такі рішення також допомагають при створенні моделей, оскільки їх можна використовувати замість складних виразів. У наведеному вище прикладі (size>5000) еквівалентно oracle_needed.

Логічні залежності між рішеннями моделюються за допомогою набору правил. Правила можна використовувати в основному для твердження, зв'язування, оновлення та інформації. Семантика правил, які використовуються для твердження та зв'язування, ідентична обмеженням, визначеним за допомогою логічних виразів у проблемах задоволення обмежень (CSP). Однак, використовуючи правила для оновлення моделі та для спілкування з користувачами під час виконання, можна вийти за рамки традиційних обмежень (оскільки це не є центром обмежень у CSP). Це також показує, що моделі варіативності на основі DoVML створюються з фокусом на інтерактивному процесі створення продукту. Ефекти рішень задаються у формі:

`if ⟨condition⟩ then ⟨action⟩`,
where `condition` is a boolean expression built using decision variables
and `action` is a function changing decision variables.

Правило атрибутується або запускається, коли його умова оцінюється як true.

2.3 Структура моделі атрибутів для контекстно-адаптивних програмних рішень

Структура та організація простору рішень є специфічними для конкретного домену, тому ядро DoVML можна параметризувати та адаптувати за допомогою мета-моделі атрибутів. Наш підхід не передбачає використання постійних типів атрибутів для моделювання варіативності. Забезпечуючи абстрактне концептуальне представлення структурованих

даних (порівняне з моделями сутності-зв'язку в реляційних базах даних), розробник визначає «мову моделювання» для простору рішень. Побудова такої моделі потребує знань про предметну область і практику впровадження в організації. Метамодель визначає типи атрибутів, які будуть включені в лінійку продуктів (наприклад, на рисунку 2.6, типами атрибутів є Компоненти, Ресурси та Властивості) і можливі зв'язки між різними типами атрибутів (на рисунку 2.6, це зв'язки «contributesTo» і «requires»).

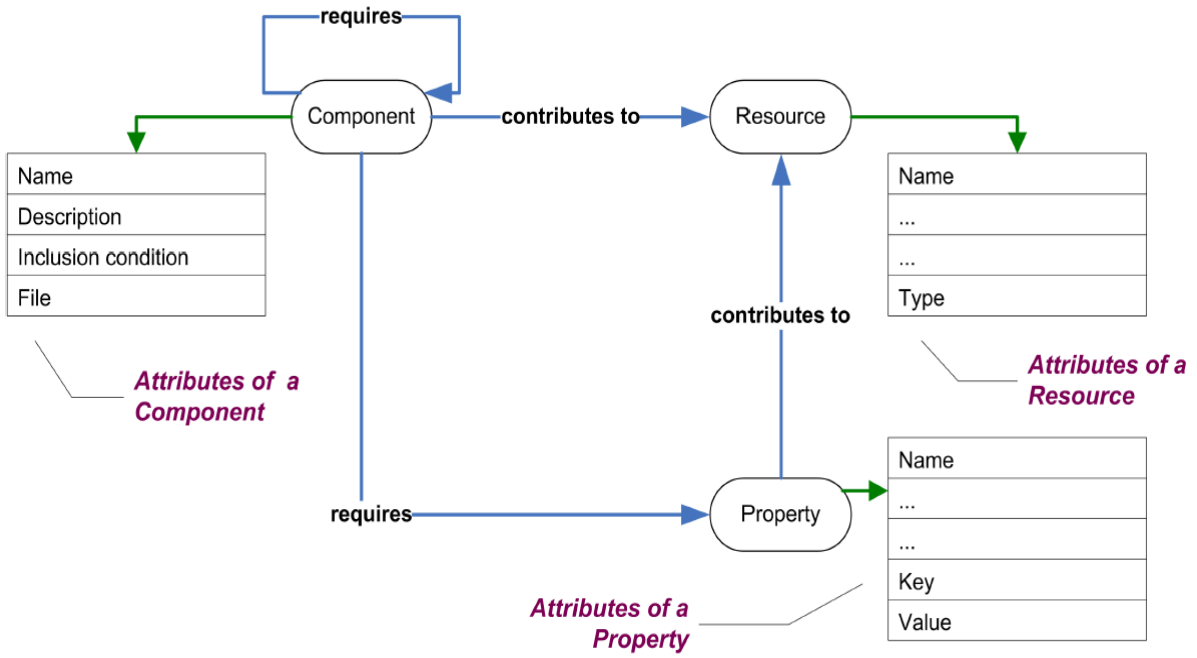


Рисунок 2.6 – Приклад (часткової) предметно-орієнтованої метамоделі, що визначає типи атрибутів та зв'язки між ними

Моделі атрибутів — це екземпляри метамоделі, що описують структуру простору рішень. Наприклад, модель атрибутів на рис 2.7 є екземпляром метамоделі, зображеної на рисунку 2.6.

Під час створення метамоделі атрибутів можна визначити їхні типи та залежності між ними.

Ми пов'язуємо логічний вираз, який називається умовою включення, до кожного атрибуту в моделі. Такий вираз визначає умову, за якої атрибут буде включений у кінцевий продукт. Якщо ресурс завжди включено в систему

(наприклад, класи корисності, загальні бібліотеки), то його умова включення є істинною. Розглядаючи приклад, представлений на рис. 2.7 умова включення компонента EMS визначається як TT3D_required. Це означає, що компонент EMS включено в кінцевий продукт, якщо для рішення TT3D_required встановлено значення true. Умова включення може бути як завгодно складною та включати будь-яку кількість змінних, таким чином підтримуючи довільно складні залежності між рішеннями та активами.

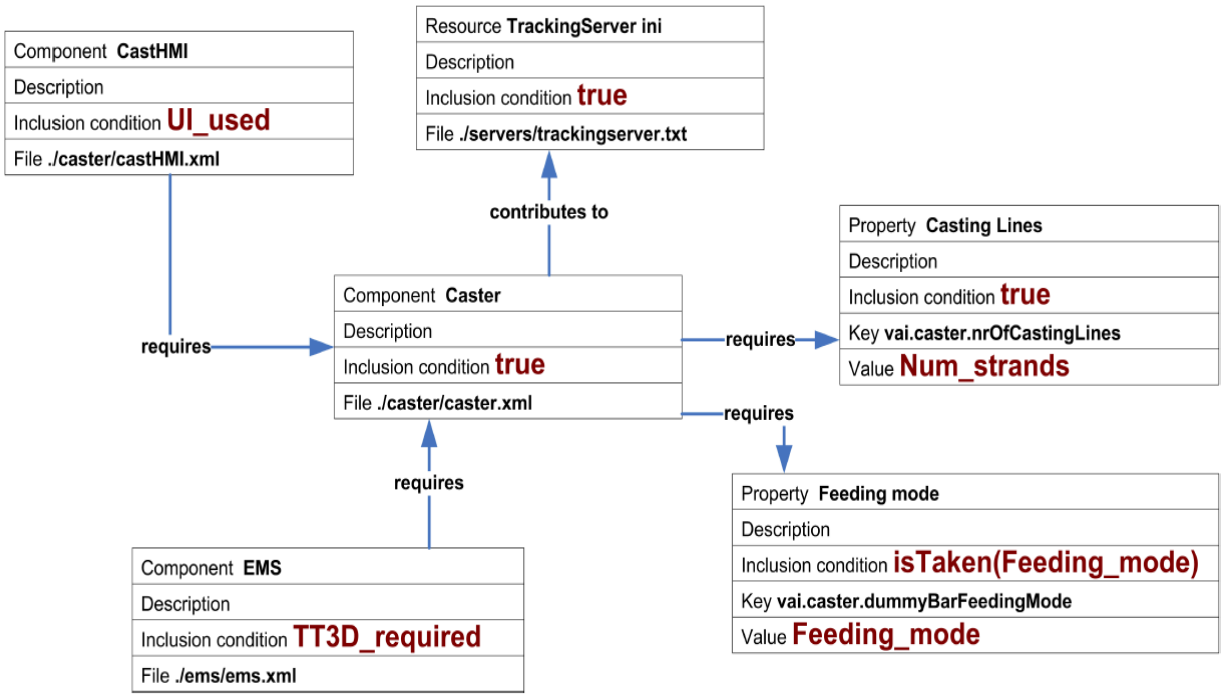


Рисунок 2.7 – Приклад часткової моделі атрибутів, що зображує їх перелік і зв’язки між ними

Часто атрибути не включаються або виключаються з кінцевого продукту безпосередньо через рішення, прийняті користувачем, а радше через технічні залежності, що є результатом їх реалізації. Наприклад на рис. 2.7, властивість Casting Lines може бути включена в кінцевий продукт, оскільки вона потрібна компоненту Caster.

Інтерпретація DoVML. Операційну семантику моделей варіативності, орієнтованих на прийняття рішень, можна пояснити за допомогою алгоритму,

який може інтерпретувати такі моделі. Результатом виконання такої моделі змінності є набір прийнятих рішень (зв’язування змінних рішень) і набір атрибутів, необхідних для бажаного продукту.

Моделі варіативності, створені за допомогою DoVML таким чином, що їх можна використовувати для високоавтоматизованих процесів отримання продуктів. Наприклад:

- умови видимості використовуються для розрізнення рішень, які є актуальними для користувача. Це залучає користувача до процесу створення продукту;
- атрибути рішення, такі як питання, описи та зображення, використовуються для інформування користувачів про рішення;
- ефекти рішення поширюються автоматично, щоб забезпечити послідовність процедури прийняття рішення.

Алгоритми виконання моделей. Прийняття рішень на основі моделей варіативності (наприклад, у вигляді виведення/конфігурації частини продукту) є інтерактивним процесом. Перехід між цими станами регулюється оцінкою умови видимості, яка запускається щоразу, коли відбувається нове зв’язування змінної. Усі видимі рішення представлені користувачеві. Зв’язування змінної відбувається в результаті взаємодії користувача або в результаті правил, які оцінюються відповідно до вимог після прийняття рішення.

Атрибут можна включити або виключити з бажаного кінцевого продукту. Перехід між цими станами відбувається в результаті оцінки стану включення атрибутів.

Псевдокод алгоритму для виконання моделі варіативності представлений у лістингу на рис. 2.8.

```
// algorithm for taking decisions
-------------------------------------------------------------------------------------------
Initialize a hash table of taken decisions <decision, value>: taken_decisions
Initialize list of required assets: required_assets

repeat {
   foreach (decision d in variability model) {
      if ( d is visible ) { // evaluation of the visibility condition
         display d.question to the user
         value val = read input from user
         if ( val is valid value of d) { // evaluation of validity condition
            add <d, value> to the table taken_decisions
            propagate the effects of decision d  // M1(d): call decision effect propagation algorithm
            calculate list of required assets // M2:call asset inclusion evaluation algorithm
         }
      }
   }
} until (all visible decisions are taken)


//M1(decision d): algorithm for propagation of decision effects
-------------------------------------------------------------------------------------------
foreach (rule r in d){
   if(r.condition evaluates to TRUE){
      execute r.script
      list l = decisions changed when executing r.script
      // propagate the effect of decision of all affected decisions
      for(each decision dx in l){
         propagate the effects of decision dx  // M1(dx): call decision effect propagation algorithm
      }
   }
}

//M2: algorithm for evaluation of asset inclusion
-------------------------------------------------------------------------------------------
foreach (Asset a in variability model){
   if(a.inclusioncondition evaluates to TRUE){
      add a to the list required_assets
      add all assets required (modeled through asset relationship) to the list required_assets
   }
}
```

Рисунок 2.8 – Приклад алгоритму для виконання моделей варіативності

По-перше, оцінюється умова видимості кожної змінної рішення. Якщо умова виконується, то користувачеві надається запитання (можливо, з іншими мітками змінної рішення), щоб змінну було краще зрозуміло під час прийняття рішення. Вхідні дані від користувача оцінюються відповідно до умови дійсності. Якщо введення було дійсним, то змінна прив’язується до вхідного значення. Таке прив’язування має два наслідки:

1. Він поширює наслідки всіх рішень, оцінюючи всі правила та виконуючи їх у міру необхідності. Такі правила також можуть викликати зв’язування змінних, що призводить до рекурсивного виклику механізму правил. Отже, виконання дії, зазначеної в правилі, вимагає, щоб умова мала

значення true. Механізм правил не використовує brute force алгоритм для оцінки всіх виразів після кожної зміни. Оцінюються лише вирази, які містять прийняте на даний момент рішення. Виконання дії може змінити набір уже зв'язаних змінних; однак також може бути лише інформативним. Оскільки механізм правил може знову запускати оцінку правил, щоб у моделі не було циклічних залежностей. Цикли в правилах виявляються за допомогою стандартних алгоритмів виявлення циклів для графоподібних структур даних.

2. Це запускає оцінку включення атрибутів, тобто процес визначення того, які атрибути потрібно включити в кінцевий продукт. Процес складається з двох етапів: оцінки умови включення і оцінки залежностей атрибутів. Потім набір включених атрибутів може використовуватися симуляторами генераторів додатків для конкретної області та інструментами розгортання для подальшої обробки.

Опишемо на прикладі, зображеному на рисунку 2.3, як модель варіативності виконується шляхом переліку одного можливого порядку прийняття рішень:

1. Перше рішення, яке може прийняти користувач - архівувати. Якщо користувач відповідає FALSE то процес завершено. В іншому випадку потрібно перейти до кроку 2.

2. Користувачеві пропонується вибрати носій рішення двома можливими варіантами є xml і db. Якщо користувач вибирає xml - перейдіть до кроку 3. Якщо користувач вибирає db, то перейдіть до кроку 4.

3. Користувач вирішує, чи потрібен очищувач .

4. Користувач вибираєє масштаб системи. Якщо масштаб більший за 5000, тоді oracle і рішення автоматично встановлюється на TRUE .

Залежно від того, які були прийняті рішення, список необхідних (включених) атрибутів розраховується шляхом оцінки умов включення та вирішення залежностей атрибутів.

Формальна семантика DoVML. Щоб описати формальне значення DoVML, потрібно дотримуватися принципів формалізації, представлених у

[22 - 23], де мова складається із синтаксичного запису, який є нескінченним набором елементів, які можуть бути використані в комунікації разом з їх семантикою. Ми використовуємо термін синтаксис щоразу, коли посилаємося на нотацію мови та зосереджуємося виключно на нотаційних аспектах мови, повністю ігноруючи будь-яке значення. Значення мови описується її семантикою. Формально синтаксис мови визначається як кортеж (T, N, P, S), де T — набір термінальних символів, N — набір нетермінальних символів, P — набір продукцій, а S — символ початку.

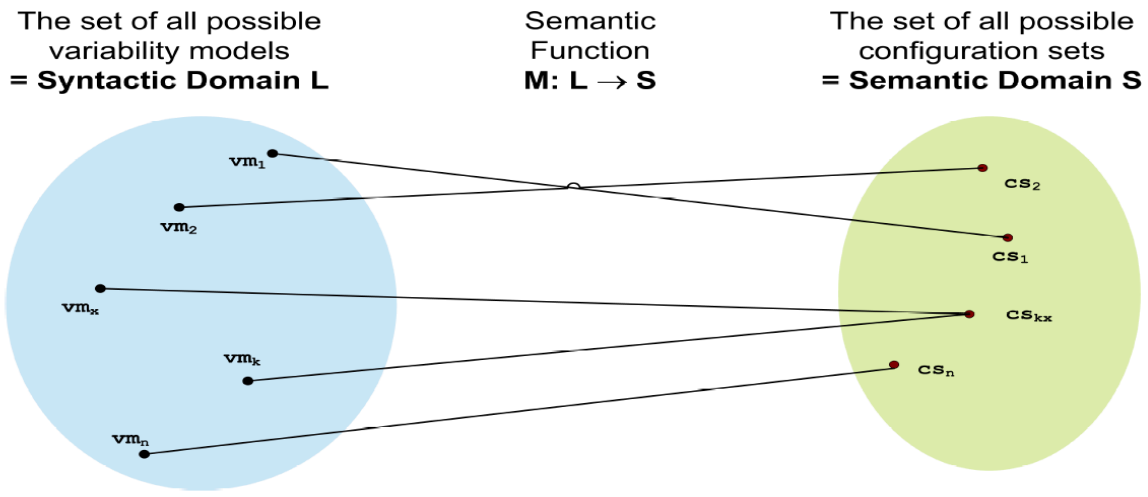


Рисунок 2.9 – Синтаксичні і семантичні домени для DoVML

Семантичне визначення для мови L складається з двох частин: семантичної області, яку ми позначаємо в загальному вигляді S і семантичного відображення синтаксису в семантичну область, позначену M. Часто відображення M пояснюється прикладами та простою мовою. Але незалежно від ступеня формальності його викладу, саме семантичне відображення має бути строго визначеною функцією від L до S, записаною M : L → S (рис. 2.8). Мова є (формально) однозначною, коли всі L, S і M отримують математичні визначення [24]. Важливо відзначити, що M є повною функцією, тобто визначеною для всіх m є L.

В DOVML складні формули створюються з рішень і простіших підформул за допомогою функцій та операцій. Щоб дати абстрактне визначення моделей рішень, немає необхідності фіксувати конкретний синтаксис, у якому розробник пише вирази, і тому припускається, що такий синтаксис існує разом із чітко визначеною семантикою.

Висновки до розділу 2

В даному розділі представлено орієнтований на прийняття рішень підхід до моделювання варіативності програмних рішень. Даний підхід описаний кількома прикладами та представлено коротку формальну семантику підходу моделювання. Варіативність моделюється через рішення, а артефакти моделюються через атрибути. Довільні типи артефактів підтримуються пропонованим підходом, оскільки основна метамодель уточнюється для кожного домену окремо. Ці артефакти можуть бути на різних рівнях деталізації та абстракції. Є два способи, якими можна підтримувати інформацію про відстеження серед довільних артефактів: в залежності від простору рішення і залежності предметної області.

РОЗДІЛ 3. ПОБУДОВА ТА ІМПЛЕМЕНТАЦІЯ МОДЕЛЕЙ ПОБУДОВИ ГНУЧКИХ ТА КОНТЕКСТНО-АДАПТИВНИХ АРХІТЕКТУР ПРОГРАМНИХ СЕРВІСІВ

3.1 Дослідження функціональності інструментів моделювання архітектур програмних сервісів

Сервіс DOPLER складається з трьох інтегрованих інструментів. Огляд функціональних можливостей, що надаються трьома інструментами, та їхніх зв’язків один з одним зображено на рисунку 3.1. Зв'язок і передача інформації між інструментами відбувається через моделі, тобто між інструментами існує залежність даних. ConfigurationWizard вимагає вихідних даних ProjectKing, який вимагає вихідних даних DecisionKing. Отже, DecisionKing є базовою платформою для моделювання, ProjectKing і ConfigurationWizard забезпечують інтерфейс для використання моделей варіативності.

DecisionKing — це інструмент для моделювання і надає функції метамоделювання для визначення предметно-орієнтованої метамоделі з типами атрибутів і взаємозалежностями між ними. Цю мета-модель потім можна використовувати для створення моделей варіативності, що стосуються предметної області.

ProjectKing — це інструмент для менеджерів продукту та реалізації попереднього налаштування моделі для продукту. Можна визначити різні ролі та обов’язки зацікавлених сторін у створенні продукту, нерелевантну змінність можна скоротити (наприклад, установивши значення за замовчуванням), а також додати додаткові знання про продажі та проект (наприклад, інформацію для підтримки прийняття рішень).

ConfigurationWizard — це інструмент для тих, хто приймає рішення (наприклад, продавців або інженерів додатків), щоб фактично приймати рішення під час розробки продукту. Інструмент надає можливості для налаштування продукту на основі моделей. Враховуються різні ролі

зацікавлених сторін і додаткові знання, створені за допомогою ProjectKing. ConfigurationWizard — це продукт, отриманий із ряду продуктів на основі прийнятих рішень.

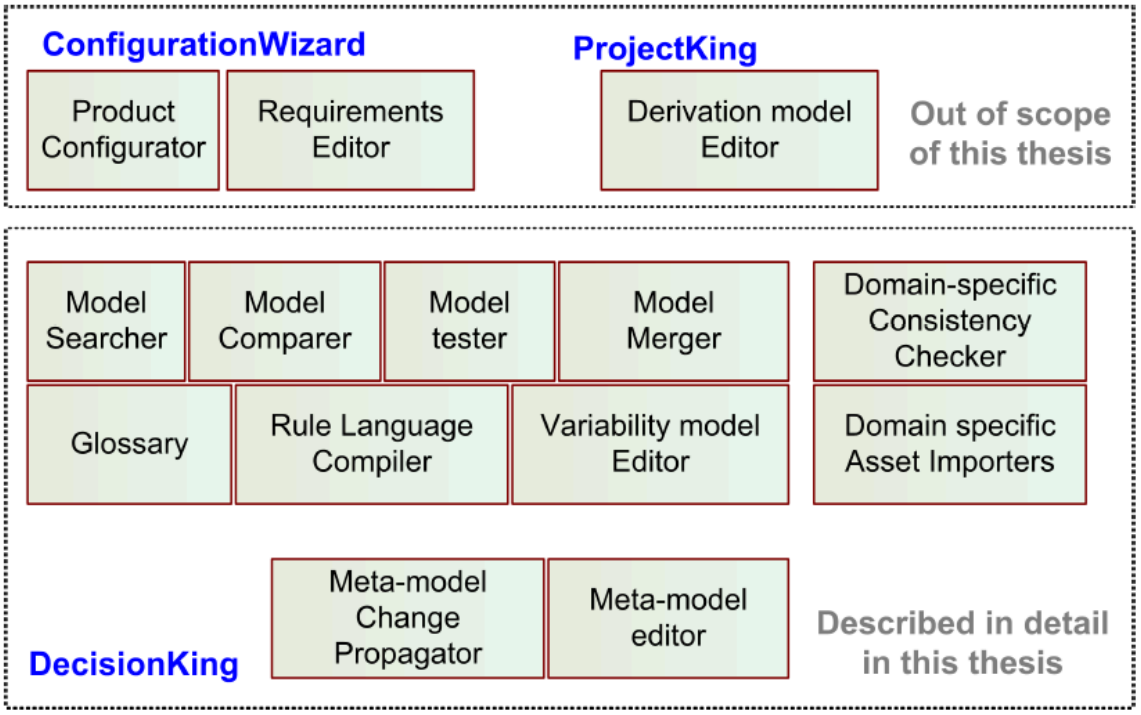


Рисунок 3.1 – Огляд інструментів DOPLER

Редактор метамоделей. Удосконалення основної метамоделі, зображеної на рисунку 2.1 можна виконати за допомогою редактора метамоделей DecisionKing. Редактор базується на Eclipse Forms і забезпечує інтерфейс користувача на основі дерева створення метамоделей. Надаючи такий інтерфейс користувача (порівняно з текстовим метамоделюванням), DecisionKing скеровує користувача під час створення метамоделі.

Інструмент надає один тип атрибутів за замовчуванням під назвою Asset, який потрібно вдосконалити для нового домену. Тип ресурсу за замовчуванням складається з трьох атрибутів за замовчуванням: Name, Description і IncludedIF.

VecisionKing реалізує такі типи: Boolean, Expression, File, List, Number, String, Paragraph та URL. Цей список можна легко розширити, надавши нові

типи атрибутів і відповідний код для створення відповідних елементів інтерфейсу користувача та пов’язаної поведінки.

На рисунку 3.2 показано редактор метамоделей у VecisionKing. Він зображує новий тип активу Component, визначений як підтип типового типу Asset. Він успадковує атрибути за замовчуванням від свого батьківського типу та визначає два нових атрибути (File і VariantType). Можна також побачити різні зв’язки між типом Компонент та іншими типами атрибутів.

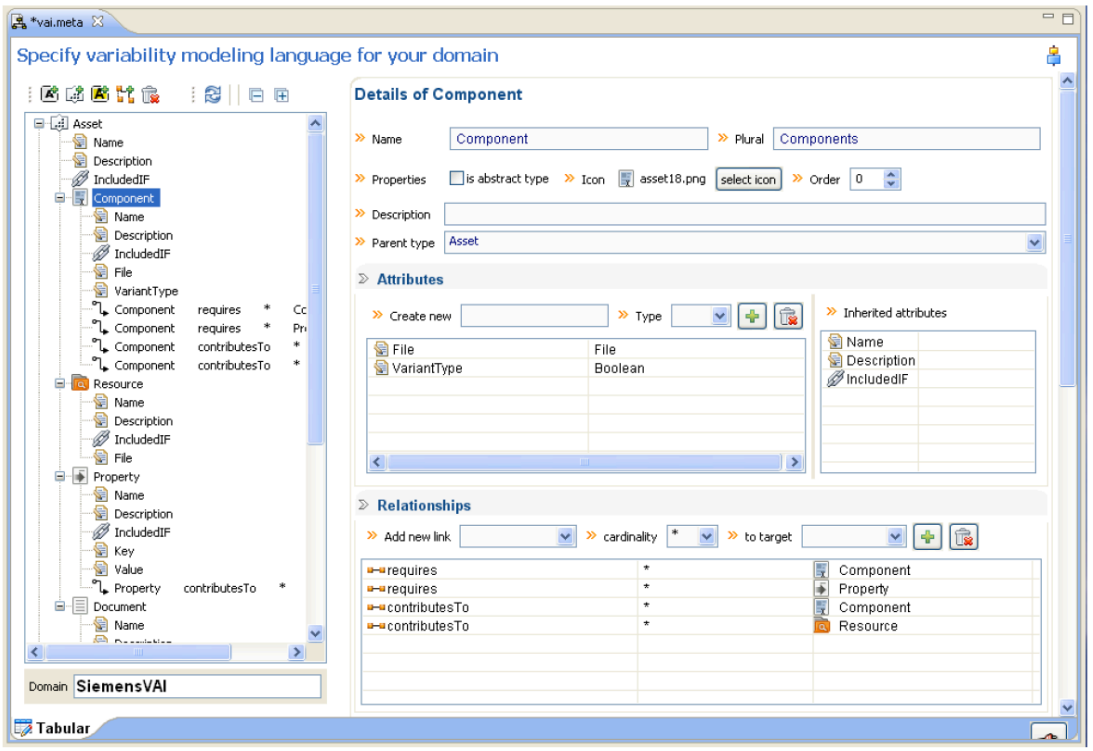


Рисунок 3.2 – Редактор метамоделі VecisionKing

Результатом роботи редактора метамоделі є файл .meta, який слугує вхідними даними для налаштування редактора моделі змінності. Потім редактор моделі дозволяє створювати моделі варіативності на основі цієї метамоделі. Подібні ідеї щодо створення предметно-орієнтованих інструментів також подані в [28], де пропонують метаінструменти, здатні генерувати домен-орієнтовані редактори візуальної мови зі специфікацій інструментів високого рівня.

Редактор моделювання. Редактор моделі варіативності – це мета-редактор для моделей, який створено для моделювання специфіки домену за допомогою файлу .meta, створеного за допомогою редактора метамоделі. Знімок екрана, зображений на рисунку 3.3 було налаштовано за допомогою метамоделі, зображеної на рисунку 3.2. Редактор моделі варіативності надає окрему сторінку моделювання для кожного типу активу.

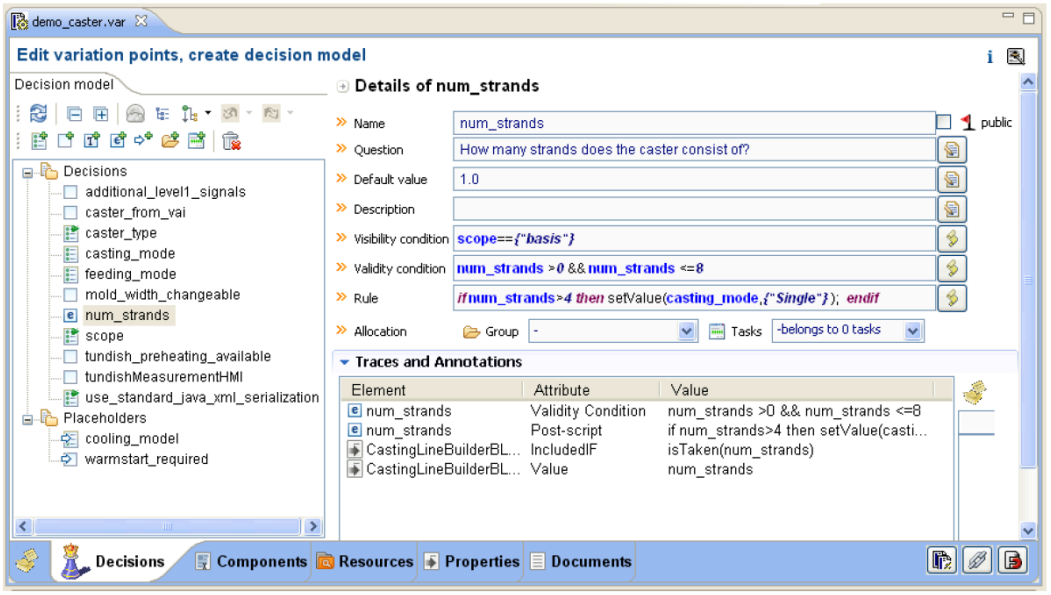


Рисунок 3.3 – Редактор моделі VecisionKing

Сторінки моделювання атрибутів дозволяють створювати атрибути відповідного типу та їх зв'язки. Таким чином, інформація, що міститься у файлі .meta, використовується для створення елементів інтерфейсу користувача, необхідних для моделювання різних типів даних.

Як показано на рисунку 3.3, редактор моделі варіативності складається з фіксованої сторінки для моделювання рішень. Наразі підтримуються три атрибути рішення для передачі значення рішення користувачу: Описи – це блоки тексту (наприклад, у HTML), які використовуються для пояснення значення рішення. HTML також дозволяє легко інтегрувати зображення, відео та анімацію для покращення керування процесом створення продукту. Питання формулюються стисло мовою проблемного простору користувача

так, щоб відповідь на це питання передбачала цінність рішення за використання анотацій до рішення можна додати довільну інформацію (у текстовому вигляді).

Інші атрибути, такі як умова видимості, умова дійсності та ефекти прийняття рішень, які потрібно формально оцінити, моделюються за допомогою мови правил та виразів і надається як плагін до VecisionKing.

Перевірка узгодженості моделей. Створення програмного забезпечення означає процес перетворення файлів вихідного коду у виконуваний код. Під час створення програмного забезпечення одним із перших кроків є компіляція вихідного коду, таким чином інструмент програмного забезпечення (компілятор) перевіряє наявність синтаксичних помилок у коді. Сьогодні автоматизовані конструктори дуже популярні в середовищах розробки програмного забезпечення. Наприклад, Eclipse надає розширені засоби створення додатків за допомогою інкрементних конструкторів проектів, які є програмами, які маніпулюють та перевіряють ресурси в проекті (моделі, код тощо) на льоту. VecisionKing використав «інфраструктуру конструктора» (надається Eclipse) і надає «конструктор варіативної моделі». Конструктор перевіряє моделі варіативності на наявність помилок і відображає виявлені невідповідності в засобі перегляду проблем (також інтегрована частина IDE Eclipse). Це дозволяє користувачам знати про різні проблеми моделювання в їхній моделі. Тут ми перелічуємо типи помилок, які зараз виявляє інструмент:

Прості правила правильного формування використовуються для виявлення деяких тривіальних помилок, які може зробити розробник, наприклад:

1. Рішення про перерахування, які мають менше двох можливих значень.
2. Обов’язкові атрибути, яким не було присвоєно значення.
3. Посилання на файли (як атрибути), що більше не існують.

Синтаксичні помилки у виразах і правилах виявляються за допомогою додатка rule-language-plugin. Логіка, необхідна для виявлення синтаксичних

помилок у виразах і правилах, залежить від мови правил і не є частиною ядра VecisionKing.

Циклічні залежності між рішеннями виявляються шляхом перетворення моделі рішень у графоподібну структуру даних. Циклічні залежності можуть виникати наступними способами:

1. Ефект рішення d1 впливає на інше рішення d2, яке у свою чергу (прямо чи опосередковано) впливає на d1.
2. Умова видимості рішення d1 залежить від рішення d2, чия умова видимості прямо чи опосередковано залежить від рішення d1.
3. Умова дійсності рішення d1 залежить від рішення d2, умова видимості якого прямо чи опосередковано залежить від рішення d1.

3.2 Процес підтримки побудови моделей

DecisionKing володіє функціональністю розширення для інфраструктури мови правил, яка складається з визначення мови, компілятора, механізму виконання та редактора [32]. Представимо систему правил які написані інтуїтивно зрозумілою мовою та переведені у відповідне представлення в нотації Drools .

Визначення мови правил складається з визначення підтримуваних типів рішень (типів даних) і функцій для маніпулювання даними. Наразі DecisionKing підтримує чотири основні типи рішень:

- логічні рішення використовуються для представлення питань «так/ні». На відміну від типу даних Boolean у мовах програмування, ми підтримуємо три можливі стани для змінних цього типу: true, false і undefined. Стан undefined був важливим, оскільки нам потрібно було розрізняти відповідь «ні» на певне запитання та «ще не вирішив»;
- числові рішення використовуються здебільшого для значень параметрів, де користувач приймає рішення щодо числового значення. Їх

можна порівняти з типом “double” у мовах програмування. Інші числові типи: цілі, короткі тощо можна моделювати за допомогою числових рішень;

- рядкові рішення використовуються для тих самих цілей, що і числові рішення. Вони відповідають типу даних «String» у мовах програмування;
- перераховані рішення можна розглядати як масиви рядків. Такі рішення використовуються щоразу, коли потрібно змоделювати різні альтернативи однієї точки варіації.

Зараз ми використовуємо мову експресії, яка демонструє високу синтаксичну подібність до Паскаля. Для побудови виразів можна використовувати стандартні оператори (наприклад +, - , ^ , * , =, =, < , > ,<, > тощо). DecisionKing надає редактор виразів (з підсвічуванням синтаксису та автоматичним завершенням (рис. 3.3), щоб полегшити процес моделювання. Окрім стандартних операторів, надаються наступні дії для запиту значення рішень і побудови складніших виразів (граматика зображена в лістингу 3.1):

1. setValue (d, p) — це функція присвоювання, яка присвоює значення p рішенню d. Функція setValue була використана замість звичайного оператора призначення « = ».
2. selectOption (d, p), deselectOption(d, p) використовуються для вибору та скасування вибору альтернативи p у рішенні про перерахування.
3. contains (d, p) – це оператор набору, який можна використовувати в рішеннях про перерахування для виконання операцій належності який порівнюється зі значенням рішення d.
4. allow (d, p), disallow(d, p) використовуються для розширення та обмеження набору можливих значень набором у рішенні про перерахування d.
5. isTaken (d) використовується для запиту, чи вже було прийняте рішення користувачем.

6. reset (d) використовується для відкликання прийнятого рішення. Відкликання рішення також скидає всі його наслідки, змодельовані в правилах.

```
RuleLangCompiler     = { Rule }.
Rule                 = ( "if" Expression "then" { Action } "endif" ) | Action.
Action               = [ ActionFunctionName "("  Parameters ")" ] ";".
Parameters           = [ Expression { "," Expression } ].
Function             = "contains" | "isTaken" | "max" | "abs".
ActionFunctionName   = "setValue" | "reset" | "selectOption" |
                       "deSelectOption" | "allow" | "disAllow".
Expression           = AndExpr { "||" AndExpr }.
AndExpr              = EqlExpr { "&&" EqlExpr }.
EqlExpr              = RelExpr { ( "==" | "!=" ) RelExpr }.
RelExpr              = AddExpr { ( "<" | ">" | "<=" | ">=" ) AndExpr }.
AddExpr              = MulExpr { ( "+" | "-" ) MulExpr }.
MulExpr              = Unary { ( "*" | "/" | "%" ) Unary}.
Unary                = {"+" | "-" | "!" } Primary.
Primary              = (Literal | ident | Function "(" Parameters ")".
Literal              = numberLiteral | stringLiteral | true | false | null.
(** END of Grammar **)
```

Рисунок 3.4 – Граматика мови правил VecisionKing, яка використовується для визначення залежностей між рішеннями

Редактор мови правил. Спеціальний редактор правил направляє користувача під час моделювання залежностей між рішеннями. Завершення коду є однією з найважливіших функцій, яка має вирішальне значення для прийняття користувачем. Завершення коду відкриває спливаюче меню зі списком змінних, які використовуються в сценарії, коли користувач починає вводити назву нової змінної. Підказки показують аргументи та значення, що повертаються для щойно введеної функції, а також короткий опис для них. Такі засоби допомагають користувачеві, оскільки не вимагають детального знання синтаксису мови правил. Редактор мови правил і підтримка автозаповнення коду зображено на рисунку 3.5.

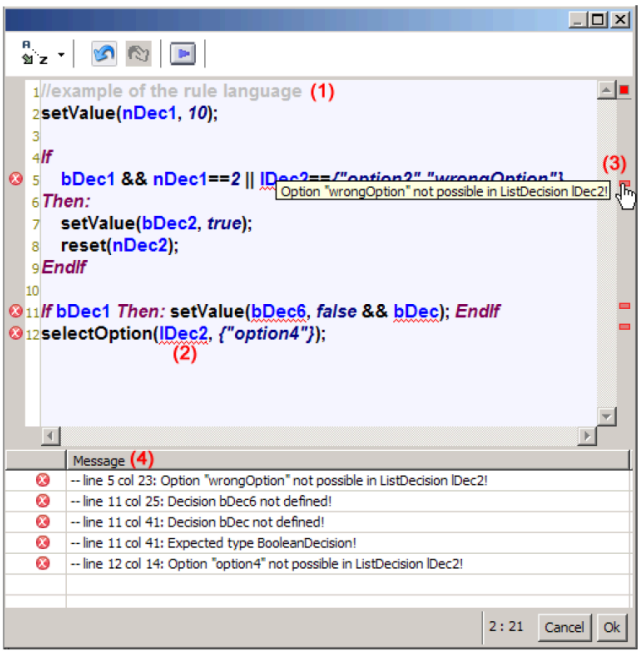


Рисунок 3.5 – Редактор мови правил

Крім того, редактор мови правил забезпечує функцію підсвічування синтаксису, що покращує читабельність сценарію. Ми також надаємо функції перевірки синтаксису на льоту, повідомляючи користувача про друкарські та інші помилки, яким можна запобігти.

На рисунку 3.4 показано типовий знімок екрана вікна редактора, де показано деякі функції, такі як (1) коментарі (2) синтаксичні помилки (3) маркери помилок і повідомлення про помилки (4) засіб перегляду помилок.

Компілятор і механізм виконання

Щоб створити компілятор для мови правил, можна використали компілятор Coco/R, який є генератором компілятора. Coco/R приймає атрибутовану граматику (ATG) вихідної мови та генерує сканер і аналізатор для цієї мови [33]. Користувач повинен надати основний клас, який викликає аналізатор, а також семантичні класи (наприклад, обробник таблиці символів або генератор коду), які використовуються семантичними діями в аналізаторі.

За допомогою компілятора, абстрактне синтаксичне дерево моделі варіативності перетворюється на структуру JBoss Drools, яка є об’єктно-орієнтованим рушієм правил з відкритим кодом. JBoss став

популярною структурою бізнес-логіки, яка використовується розробниками Java для створення складних програм на основі правил шляхом поєднання платформи Java і технології бізнес-правил. Його базова архітектура відповідає структурі класичної експертної системи.

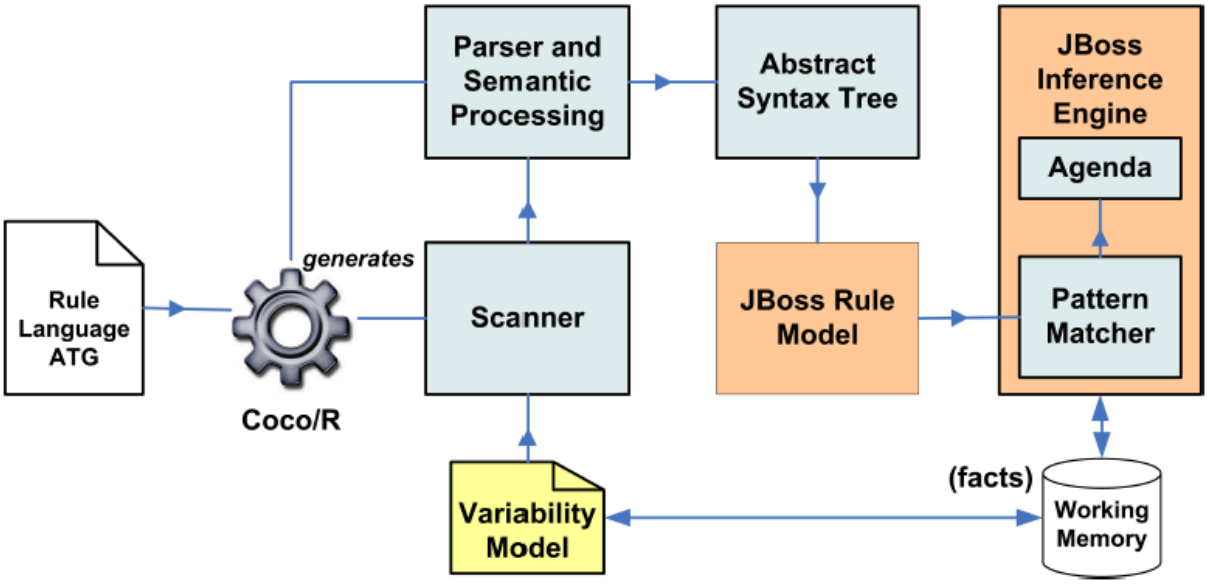


Рисунок 3.6 – Генерація компілятора мови правил і оцінка моделей за допомогою JBoss Rule Execution Engine

Рисунок 3.6 показує, як компілятор мови правил інтегрується в механізм виконання правил, наданого JBoss Drools. Концепція експертної системи така: знання експерта закодовані в наборі правил і при роботі з тими самими даними експертна система працюватиме так само, як експерт.

JBoss Drools — це система правил прямого ланцюжка. Він починається з бази правил, яка містить усі відповідні знання, закодовані в правилах If-Then і робочої пам'яті, яка спочатку може містити або не містити будь-які дані, твердження або інформацію. У випадку DecisionKing правила – це ефекти рішення, змодельовані як атрибути рішення, а факти — це рішення, прийняті користувачем. Система перевіряє всі умови правила та визначає підмножину правил, умови яких задовольняються на основі робочої пам'яті. Цей процес називається «Відповідність шаблону» і базується на алгоритмі

Rete [34], який оцінює декларативний предикат зі змінним набором правил у реальному часі. JBoss Drools обертає семантику звичайного реляційного Rete в ReteOO (об'єктно-орієнтовану) модель, яка більш сумісна з об'єктами Java. Коли правило запускається, виконуються будь-які дії, зазначені в пункті THEN. Ці дії можуть змінювати робочу пам'ять, саму базу правил або виконувати будь-яку операцію. Цей цикл запуску правил і виконання дій продовжується, доки не буде виконано одну з двох умов: більше немає правил, умови яких задовольняються або запускається правило, дія якого вказує, що програму слід завершити.

Щоб зрозуміти синтаксис JBoss Drools, представимо приклад правила, написаного мовою правил DecisionKing і його автоматичний переклад у нотацію JBoss Drools (рис. 3.7).

```
//The following is a simple rule consisting of one if statement:
if (num_strands>4)then
  setValue(casting_mode, {"Single"});
endif

//is automatically translated into drools rule in the following form:
rule "0"
salience 0
no-loop true
when
  num_strands:RuleNumberDecision(
    name == "num_strands",
    active==true)and
    eval(num_strands.getPValue()>4)then
      ArrayList<String> drools_a;
      num_strands.identify();
      drools_a = new ArrayList<String>();
      drools_a.add("Single");
      num_strands.set(0,"casting_mode", drools_a);
  end
```

Рисунок 3.7 – Приклад правила в DecisionKing і його перетворення в нотацію Drools

Ось простий приклад того, як Rete оптимізує мережу правил і чому він швидший ніж метод пакетного виконання правил. Дано два правила, як на

рисунку 3.8, існують різні способи, як можна оцінити правила. Як показано на рисунку 3.9, JBoss Rule Engine, прийнятий DecisionKing, оптимізує дерево Rete таким чином, що оцінка виразів є ефективнішою, оскільки щоразу, коли змінюється рішення, потрібно оцінювати лише вибрану підмножину виразів.

```
//rule 1:
if (a && b) then setValue(d, true);

//rule 2:
if (a && b && c) then setValue(e, true);
```

Рисунок 3.8 – Приклади двох простих правил у DecisionKing

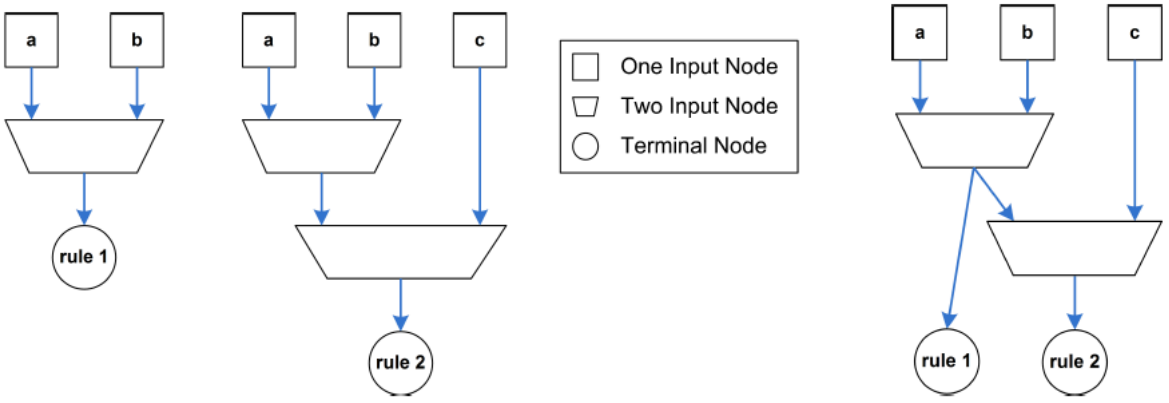


Рисунок 3.9 – Приклад мережі Rete оптимізованої JBoss Rule Engine для легкої оцінки при зміні рішення

3.3 Підхід до моделювання гнучких та контекстно-адаптивних архітектур програмного забезпечення

В даному пункті пропонується багатокомпонентний підхід для структурування простору моделювання, який дозволяє працювати з меншими моделями, таким чином зменшуючи складність їх створення та обслуговування. Зміни можуть бути внесені локально в окремі фрагменти

моделі призначеною групою експертів. Огляд пропонованого підходу зображено на рисунку 3.10.

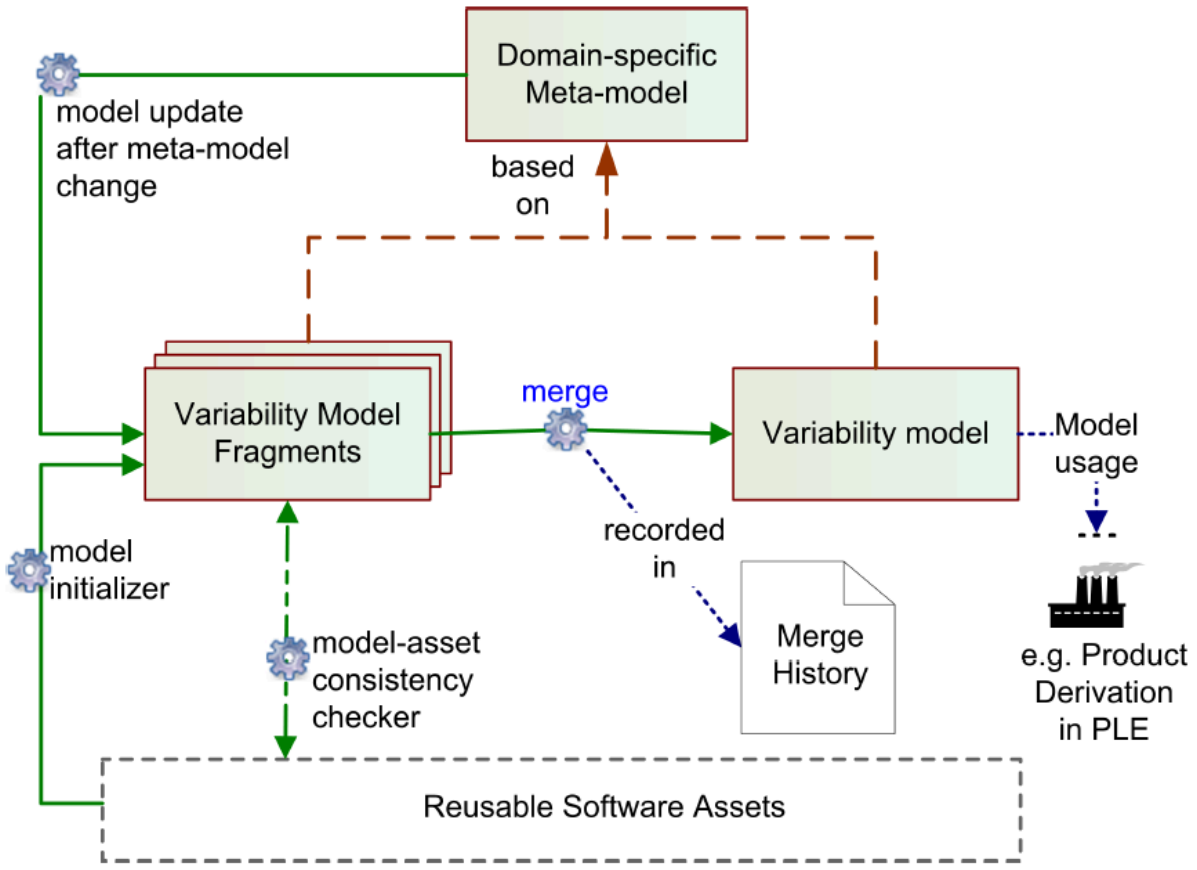


Рисунок 3.10 – Огляд багатокомпонентного підходу на основі фрагментів моделі

Ключові елементи підходу наступні. Фрагмент моделі описує повторно використовувані атрибути та їх варіабельність довільної частини ряду продуктів (наприклад, набору функцій, підсистеми або наскрізної функціональності). Фрагменти моделі служать основною одиницею еволюції та створюються та підтримуються окремими зацікавленими сторонами лише у слабкому зв’язку з діяльністю інших зацікавлених сторін. Фрагменти моделі ніколи не використовуються безпосередньо для отримання продукту.

Модель варіативності об’єднується з набору фрагментів моделі в певних точках життєвого циклу лінійки продуктів (наприклад, перед

початком розробки продукту). На відміну від фрагментів моделі, модель варіативності може бути використана для отримання продукту [37], оскільки всі невідповідності між складовими фрагментами були вирішені під час злиття. Отриману модель не можна змінювати, оскільки вона оновлюється шляхом повторного злиття фрагментів моделі.

Під час злиття встановлюються зв’язки між фрагментами моделі та результатом процесу злиття. Власники фрагментів моделі використовують історію злиття, щоб переглядати свої окремі фрагменти на основі застосованих дій вирішення конфліктів, щоб прискорити майбутні процеси злиття.

Фрагменти моделі змінності базуються на предметно-орієнтованій мета-моделі, яка визначає специфіку організації або домену шляхом визначення типів атрибутів, які мають повторно використовуватися в лінійці продуктів разом із їхніми атрибутами та залежностями. Таким чином, модель результату процесу злиття також базується на тій самій предметно-орієнтованій метамоделі. Моделі варіативності потрібно оновлювати автоматично після змін у метамоделі.

Фрагменти моделі. Фрагмент моделі складається з двох типів сутностей моделювання (нижня половина рисунка 3.8): елементи моделі та елементи - заповнювачі. Було взято концепції з об’єктно-орієнтованих мов програмування для визначення видимості елементів моделі у фрагментах моделі. Подібні до приватних і публічних елементів у класах, розробники можуть вказати загальнодоступні елементи фрагмента, щоб зробити їх видимими за межами моделі (рис. 3.11) . Елементи моделі визначаються як приватні елементи, якщо вони не мають відношення до інших частин системи або не повинні бути відомі зовні для моделювання змінності, тобто вони є внутрішніми для підсистеми і не мають прямих зв’язків з елементами інших моделей.

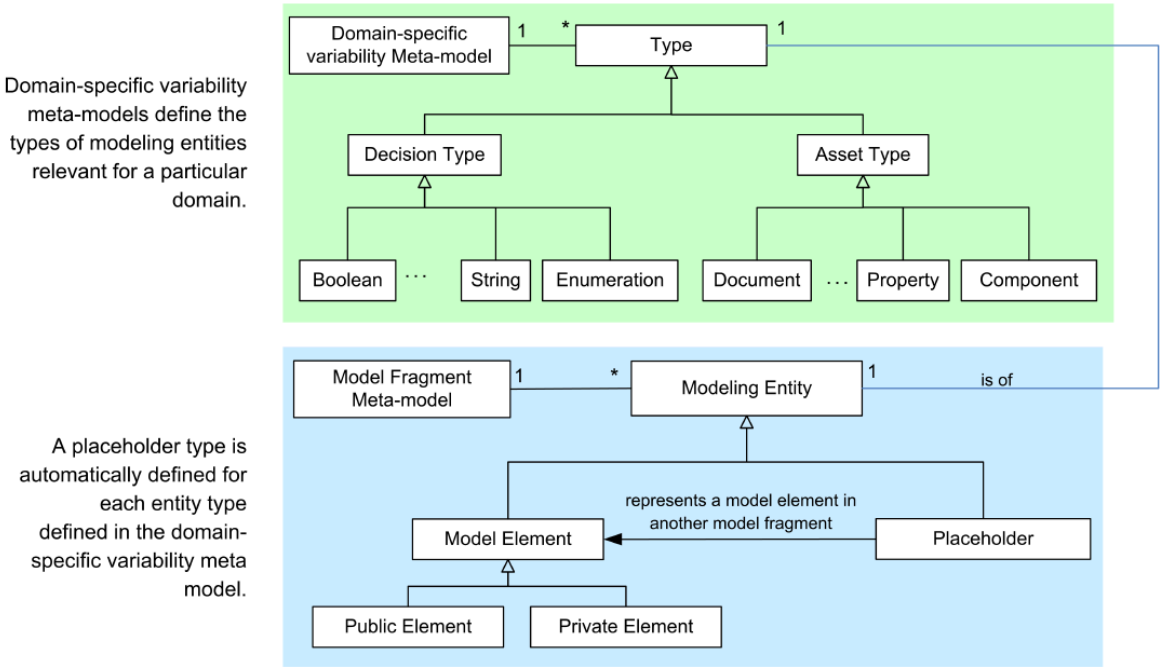


Рисунок 3.11 –- Метамодель високого рівня, що зображує різні моделі та їхні залежності

Елементи-заповнювачі можна порівняти з необхідними інтерфейсами в мові програмування. Вони мають тип даних, визначений у метамоделі і їх можна розглядати як теговані та типізовані змінні, які замінюються елементами моделі в інших фрагментах моделі під час злиття. Елементи-заповнювачі вводяться у фрагмент моделі щоразу, коли потрібно визначити зв’язки з елементами з інших фрагментів моделі. Це, наприклад, необхідно під час визначення правил композиції продукту між елементами. Явне розташування або точні назви елементів, на які посилаються, не потрібні під час моделювання, щоб дозволити слабкі зв’язки між фрагментами [39].

Приклад: фрагмент моделі 1 на рисунку 3.12 містить заповнювач для архіву рішень, який використовується для визначення залежності між dbSupport і архівом. Рішення dbRequired у фрагменті моделі 2 є заповнювачем для рішення, визначеного в іншому фрагменті моделі. Щоб полегшити еволюцію, при створенні фрагменту 2, не потрібно знати справжню назву

елемента dbSupport. Під час злиття dbRequired замінюється на dbSupport із фрагмента моделі 1, щоб вирішити цю неоднозначність.

Створюється довільна кількість фрагментів моделі для різних частин системи на основі предметно-орієнтованої метамоделі. Фрагменти моделі можуть розвиватися незалежно та з різною швидкістю.

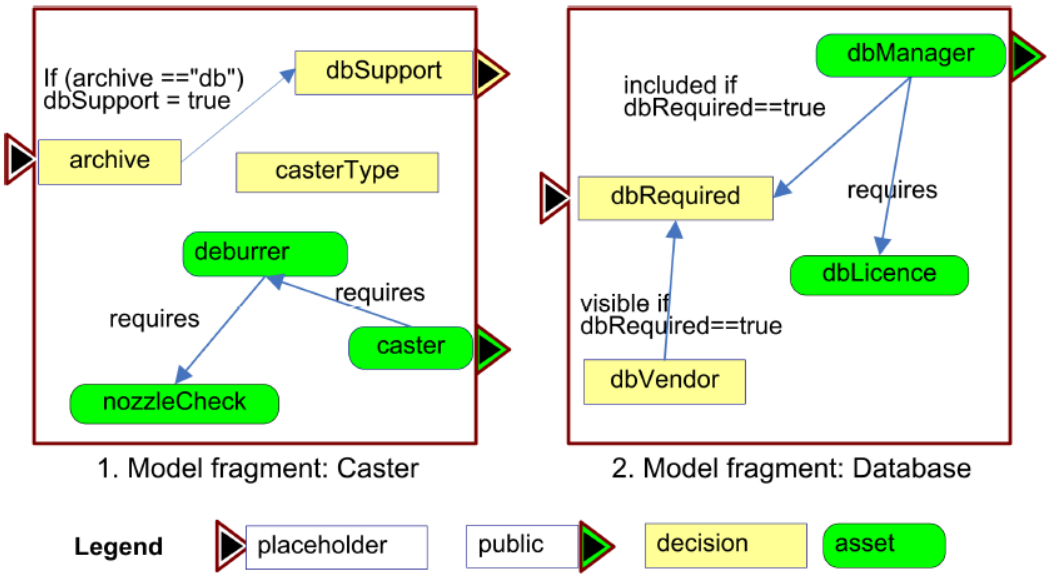


Рисунок 3.12 – Приклад фрагментів моделі

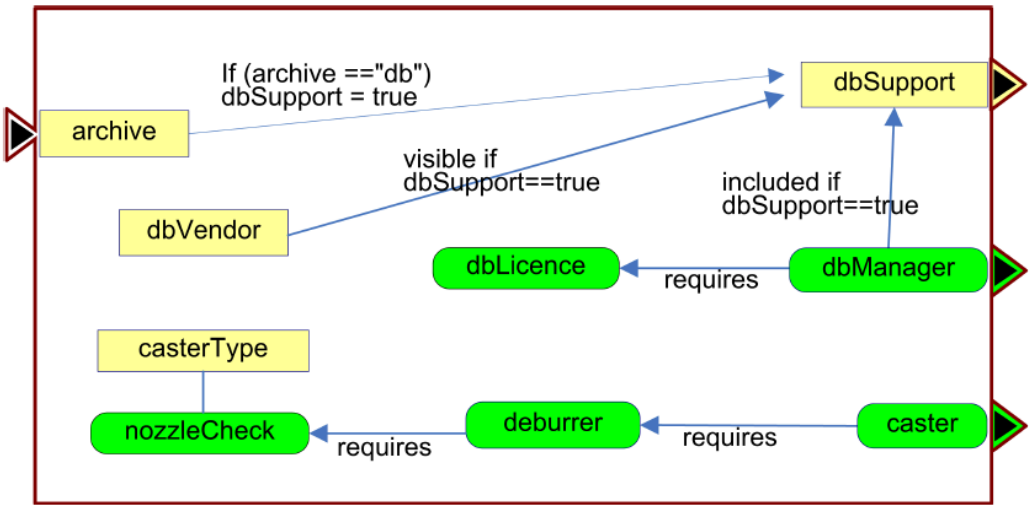


Рисунок 3.13 – Результат злиття двох фрагментів, зображених на рисунку 3.11.

За потреби інструменти напівавтоматично об’єднують фрагменти в єдину модель змінності. У прикладі, зображеному на рис 3.14 розробник створює єдину модель у момент часу t1 на основі версій фрагментів моделі, доступних у цей час.

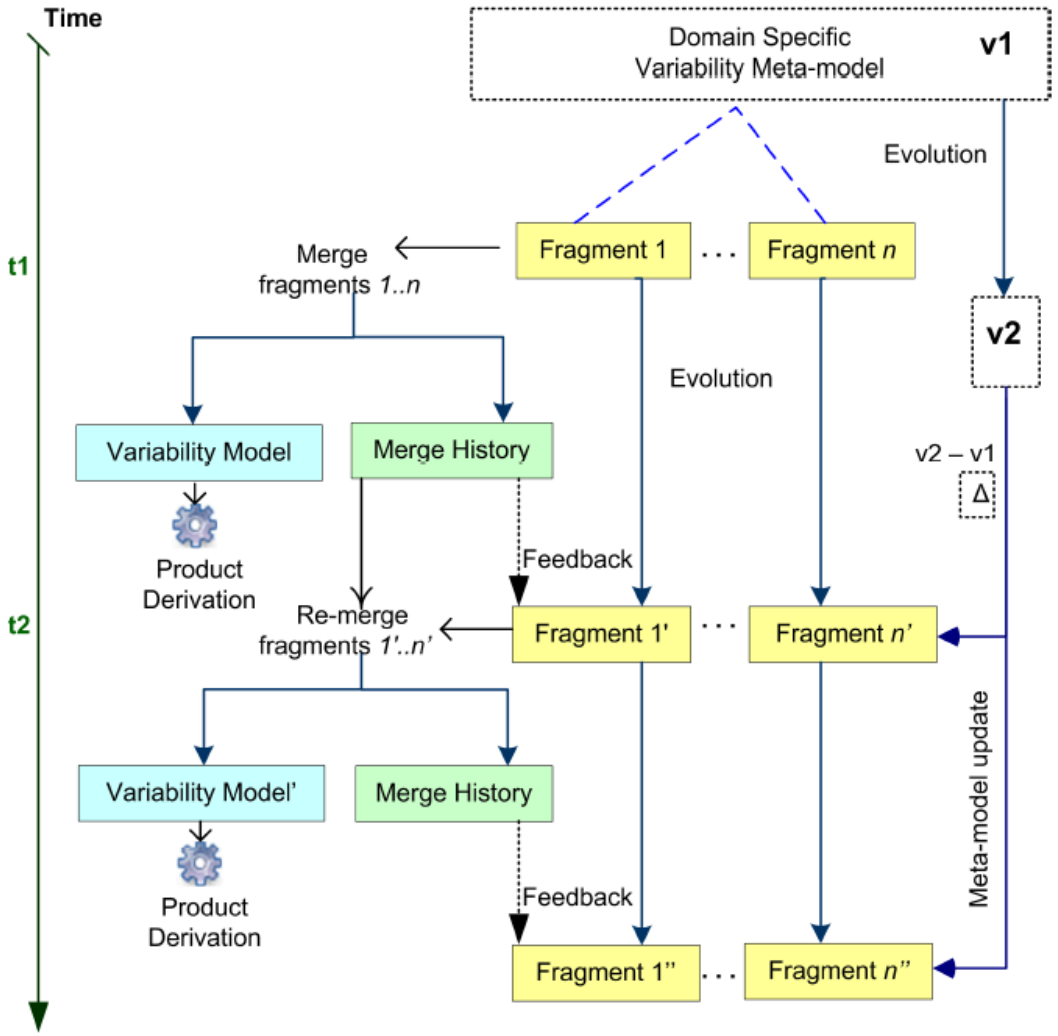


Рисунок 3.14 – Еволюція моделі відбувається на моделі варіативності та відповідному рівні метамоделі

Керівники проектів можуть використовувати об’єднану модель для отримання продукту. Об’єднану модель не можна змінювати під час створення. Після набору змін до різних фрагментів у момент часу t2 ініціюється інший процес злиття. Повторне злиття отримує переваги від вирішення конфліктів у злитті в t1. Об’єднану модель можна використовувати

для отримання іншого продукту. У прикладі предметно-орієнтована метамодель змінюється з v1 на v2 . Це вимагатиме оновлення існуючих фрагментів для забезпечення узгодження з новою метамоделлю v2.

Злиття фрагментів. Варто зазначити, що декомпозиція передбачає рекомпозицію [39], що означає, що робота з малими моделями потребує техніки для фактичного створення великої моделі з маленьких. Фрагменти моделі є неповними, оскільки вони представляють лише часткове уявлення про систему. Зв’язки з іншими частинами системи, змодельованими за допомогою покажчиків місця заповнення, потрібно вирішити, перш ніж можна буде створити єдину модель для похідного продукту. Під час злиття елементи складових фрагментів моделі збираються в нову модель, а заповнювачі замінюються відповідними елементами моделі з інших фрагментів моделі. Важливо відзначити, що фрагменти вихідної моделі залишаються незмінними під час злиття.

Подібні підходи до спільного моделювання ознак були представлені в [40], де автори представляють процес незаблокованого багатоклієнтського спільного моделювання функцій, що складається з методу коригування функцій для вирішення конфліктів і розширеного механізму іменування для збереження намірів проекту. У таблиці 3.1 наведено чотири типи конфліктів злиття разом із вирішенням.

Таблиця 3.1

Опис різних типів конфліктів злиття та можливе їх вирішення

Merge conflict	**Resolution strategy**
Multiple occurrences of same identifier	Rename involved elements or drop all others but one
Name mismatches	Synonym check with glossary Rename one of the mismatching elements
Multiple definitions	User confirm semantic equality Delete one of the instances
No matching elements for place-holders	Automatically suggest a candidate resolution. If no matches are found, placeholders remain unchanged

Стратегія злиття моделі подана на рисунку 3.15.

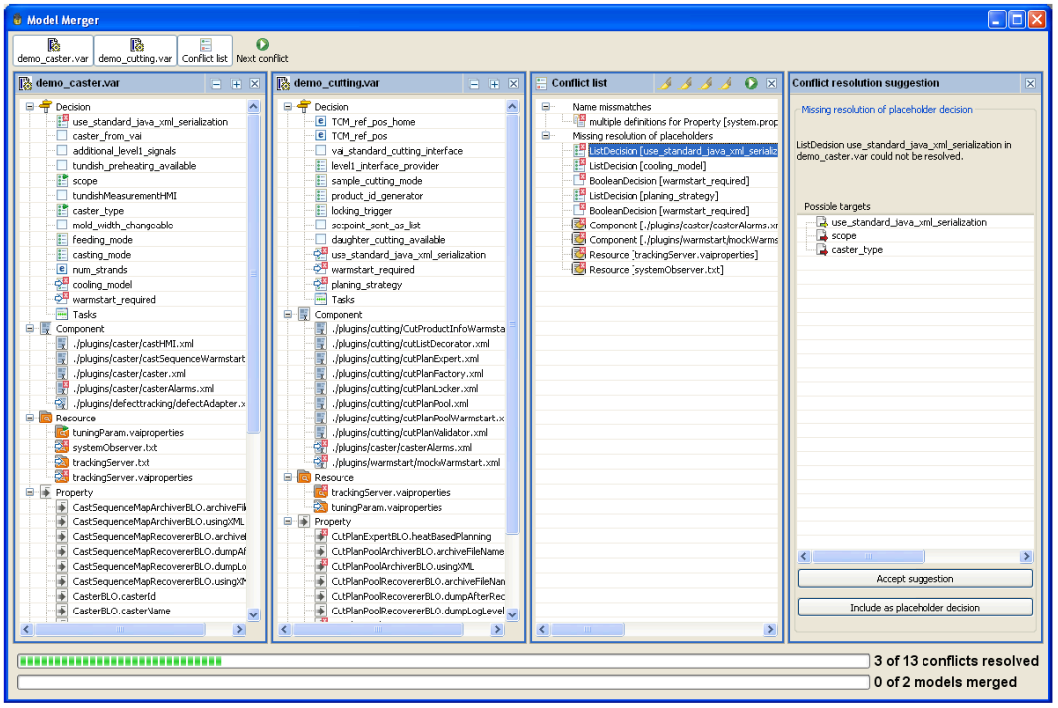


Рисунок 3.15 – Злиття моделі варіативності DecisionKing

Важливо, щоб усі елементи моделі мали унікальний ідентифікатор. Однак, оскільки розробники моделі не знають про інші фрагменти моделі під час моделювання, елементи в різних фрагментах моделі можуть мати однакові назви. Це призводить до конфлікту під час злиття. Принаймні один із конфліктуючих елементів має бути перейменовано або видалено.

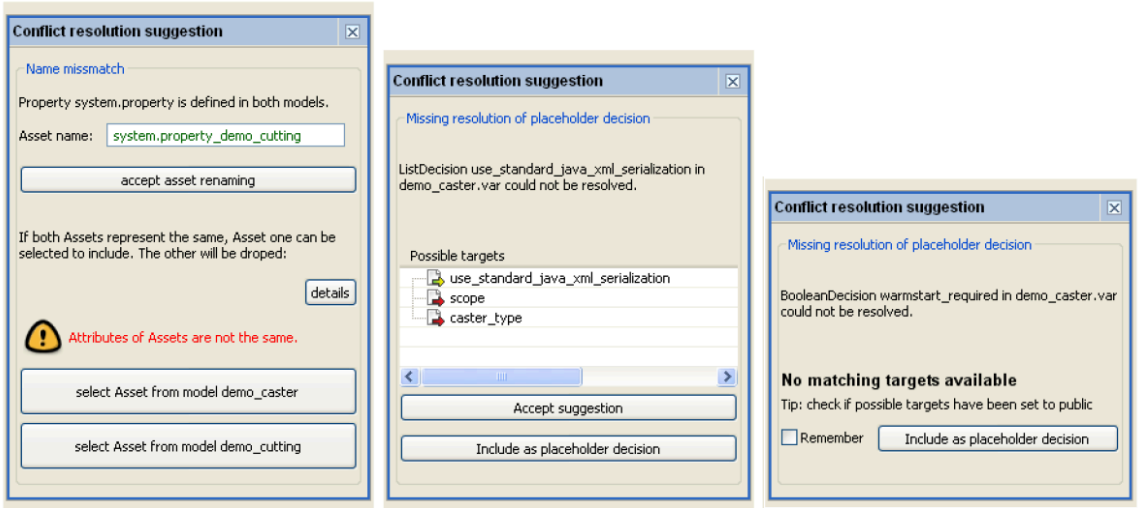


Рисунок 3.16 – Пропозиції злиття для вирішення конфліктів

Невідповідність імен. Ім'я заповнювача може не збігатися з ім'ям призначеного елемента. Наприклад, фрагмент моделі може містити елемент моделі з ім’ям admin, а деякі інші фрагменти можуть посилатися на той самий елемент за допомогою імені sysAdmin. Такі випадки важко розв’язати повністю автоматично, і ми покладаємося на експерта-людина під час об’єднання, щоб підтвердити семантичну рівність використаних назв елементів. Однак інструменти використовують глосарії для певних доменів, які визначають синоніми використаних імен, щоб полегшити об’єднання в таких випадках.

Кілька визначень. Різні фрагменти моделі можуть визначати зміну загальної частини системи. Це може статися, наприклад, коли спільні компоненти використовуються більш ніж однією підсистемою, і кілька власників підсистеми вирішують моделювати мінливість спільних компонентів як частину своєї підсистеми.

Тоді алгоритм виявляє всі екземпляри елементів (на основі правил іменування, типів елементів і зв’язків між елементами) і включає лише один екземпляр у об’єднану модель. Наприклад, щоразу, коли компонент з однаковою назвою міститься в більш ніж одному фрагменті моделі, користувач вирішує:

- перейменувати один із компонентів перед об’єднанням;
- включити компонент лише один раз до об’єднаної моделі.

Немає відповідних елементів для заповнювачів. Також можливо, що жоден модельєр не відчуває відповідальності за певну частину системи. У результаті кілька фрагментів моделі можуть визначати заповнювачі, для яких не існує реального елемента моделі. Для вирішення проблеми знову потрібне втручання користувача. Користувач вибирає зв'язувальний елемент зі списку запропонованих елементів-кандидатів. Якщо елемент прив’язки недоступний, то результуюча модель варіативності все ще міститиме невирішені заповнювачі.

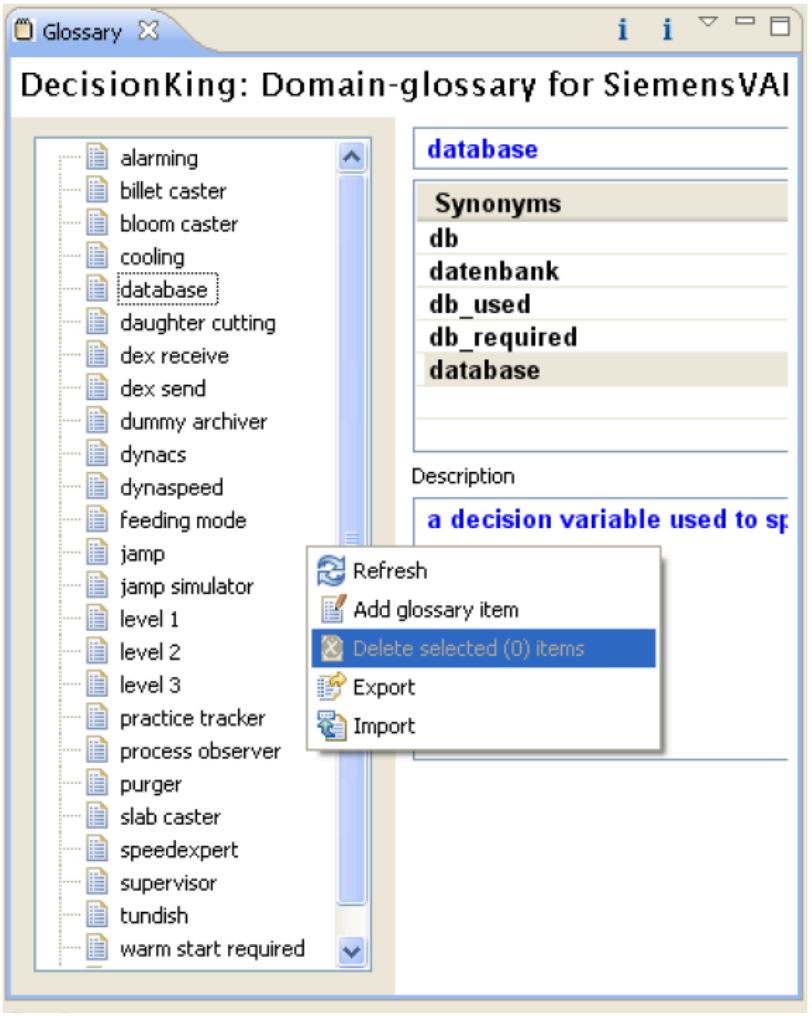


Рисунок 3.17 – Інструмент глосарію домену VecisionKing для перевірки синонімів під час злиття

Приклад. На рисунку 3.13 зображено результат об’єднання фрагментів моделі з рисунку 3.12. В ідеалі всі еталонні елементи збігаються з публічним елементом в інших моделях. Однак, оскільки фрагменти моделі спочатку створюються без явного узгодження та базуються на "вільних посиланнях", під час злиття можуть виникати різні конфлікти. Кожного разу, коли елементи моделі перейменовуються (у разі конфлікту імен), видаляються або видаляються (у разі кількох випадків), їх атрибути, обмеження та умови також потрібно оновити відповідним чином. Оскільки елемент заповнювача dbRequired було зіставлено з dbSupport, наприклад, така умова (у фрагменті моделі 2 на рисунку 3.12):

```
visibleif dbRequired==true
```

буде автоматично змінено на наступний стан під час процесу злиття (рис 3.13):

```
visibleif dbSupport==true
```

Під час процесу злиття моделі ми записуємо застосовані зміни та прив’язки в історію злиття (рис. 3.18).

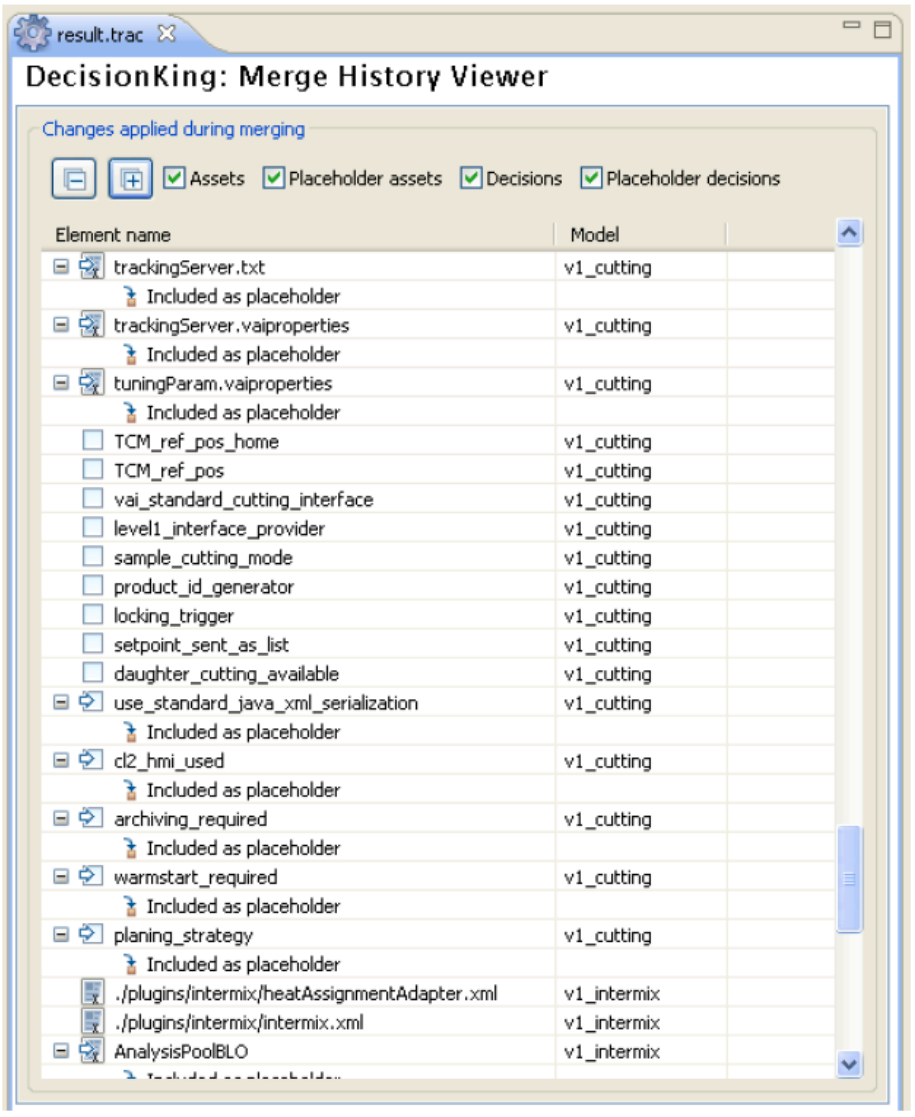


Рисунок 3.18 – Інструмент перегляду історії злиття VecisionKing

Зміни можуть бути двох типів, тобто зміни, зроблені автоматично інструментом об’єднання і зміни зроблено явно користувачем. Історія злиття має три важливі функції:

Простежуваність вперед і назад. Історія злиття пов’язує фрагменти моделі та модель змін для підтримки прямої відстежуваності - «як мій фрагмент моделі використовується в моделі змін ?» і зворотної відстежуваності - «з якого фрагмента моделі походить певний елемент у моделі змін ?».

Зворотній зв'язок з власниками фрагментів моделі. Вихідні фрагменти моделі залишаються незмінними після злиття. Однак власники фрагментів моделі інформуються про зміни вирішення конфліктів, такі як видалення елементів, посилань і зв’язків, застосованих під час злиття, щоб уникнути повільного відключення модельєрів один від одного. Розробники моделей отримують інформацію щодо дій зі змінами, які були необхідні під час злиття (наприклад, які елементи потрібно було перейменувати). Вони можуть вирішити переглянути фрагменти своєї моделі на основі цього відгуку (табл 3.2) . Це допомагає розробникам моделей об’єднати та узгодити визначення у фрагментах моделі.

Таблиця 3.2

Різні типи стратегій злиття та можливі відгуки

Merge strategy	Feedback to fragment owner
Element renamed	Rename element in the fragment to ensure positive match
Element deleted	Either drop element from fragment or change element to a placeholder
Missing resolutions of reference	Either remove the reference element or change from reference to definition type

Повторюваність злиття. У разі частих змін фрагментів і великої кількості фрагментів повторення процесу злиття кожного разу з нуля може бути тривалим. Кожного разу, коли процес злиття потрібно повторити після

внесення змін до фрагментів моделі, історія злиття використовується для відтворення попередньо виконаних дій змін із мінімальним втручанням користувача (швидке повторне злиття). Окрім вибору користувача під час об’єднання, історія злиття містить усі дії змін, які автоматично виконує інструмент об’єднання (наприклад, перейменування, відображення посилань або зміни в атрибутах чи зв’язках).

3.4 Перевірка відповідності та адекватності побудови моделей

Основні атрибути лінійки продуктів постійно розвиваються, щоб відповідати змінам, таким як нові вимоги клієнтів, технологічні зміни або необхідний рефакторинг. Наприклад, великий компонент може бути розділений, компонент може бути переміщений в іншу підсистему або можуть бути встановлені нові зв’язки між компонентами. Тому важливо розуміти, моделювати та підтримувати зв’язки між моделями варіативності продуктової лінії та базою атрибутів.Зусилля, необхідні для створення моделі архітектури лінійки продуктів можна мінімізувати за допомогою інструментів, які аналізують наявні атрибути та автоматично створюють початкову модель. Залежно від механізмів реалізації змінності існують різні способи ідентифікації точок варіації та змінних атрибутів. Наприклад, конструкції варіативності на рівні реалізації (такі як успадкування або параметризація) можна ідентифікувати шляхом аналізу існуючої реалізації. Існує інструмент, який автоматично генерує початкову модель варіативності лінійки продуктів для нашого галузевого партнера, аналізуючи файли конфігурації Spring XML існуючої системи.

Моделі лінійки продуктів повинні підтримуватися відповідно до архітектури під час обслуговування та еволюції. Інженерам необхідно часто змінювати архітектуру та фрагменти моделі варіативності (наприклад, при введенні нових варіантів). Невідповідності, що є результатом таких змін, потрібно автоматично виявляти та виправляти, щоб підтримувати спільну

еволюцію архітектури з відповідними фрагментами моделі та навпаки. Необхідно поширювати зміни в програмних компонентах архітектури лінійки продуктів до моделей. Подібним чином моделі не слід просто змінювати без внесення відповідних змін до архітектури. Було розроблено інструменти для автоматичного виявлення змін, які надають розробникам та інженерам лінійки продуктів миттєвий відгук про невідповідності. Виявлено, що в багатьох випадках можна відстежувати зміни в базових основних атрибутах і автоматично синхронізувати моделі, оскільки елементи в моделях змінності безпосередньо відображаються на існуючі атрибути. Наразі даний інструмент може виявити два типи невідповідностей: втрачені елементи моделі та посилання. Модель може містити елементи, які були видалені або перейменовані в базі атрибутів. Щоб усунути цю невідповідність, або застарілий елемент моделі видаляється, або база атрибутів змінюється відповідно до моделі. Модель також може містити залежності між елементами, які більше не є правильними або недоступними.

Відсутні елементи моделі та посилання. Також можливо, що не всі частини системи вже охоплено в моделі. Такі випадки трапляються, наприклад, якщо компоненти додаються до бази атрибутів або модифікуються для задоволення потреб щоденного бізнесу. Використовувані інструменти автоматично виявляють такі невідповідності та повідомляють про них користувачам.

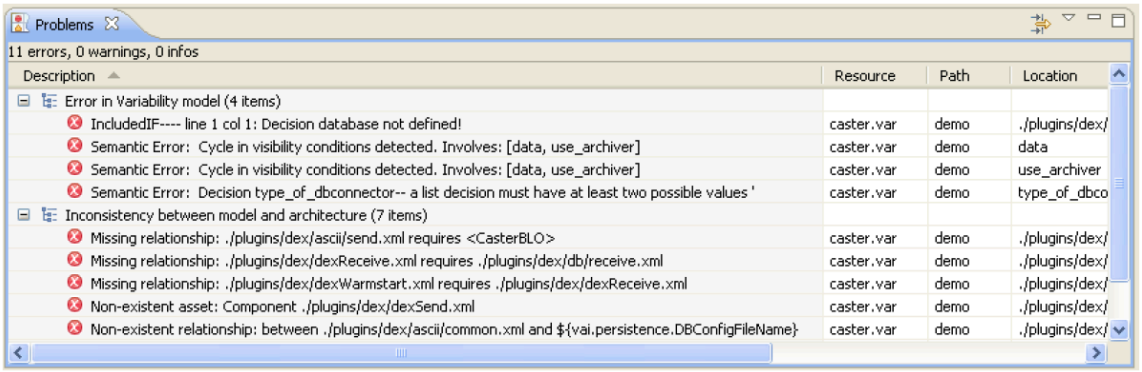


Рисунок 3.19 – Засіб перегляду проблем VecisionKing, що показує різні типи невідповідностей, виявлені інструментом синхронізації моделі-архітектури

На рисунку 3.19 показано засіб перегляду помилок в інструменті моделювання VecisionKing, який відображає невідповідності між моделлю та архітектурою. Інструмент використовує вже існуючі моделі та артефакти для пошуку невідповідностей: кожного разу, коли моделі змінюються, існуючі архітектурні елементи використовуються як еталон для порівняння. Щоразу, коли архітектурні елементи змінюються, існуючі моделі служать таблицею пошуку. Наприклад, коли новий варіант вводиться шляхом зміни моделі варіативності, інструмент гарантує, що існує артефакт із такою ж назвою та структурою (разом із залежностями від інших артефактів). Подібним чином, коли до архітектури додається новий компонент, інструмент автоматично шукає його існування в доступних моделях.

Застосування фрагментів моделі. Незалежно від того, який підхід моделювання дотримується, розробка однієї моделі лінійки продуктів практично неможлива через розмір і складність сучасних систем. Велика кількість функцій і компонентів у системах реального світу означає, що розробникам моделей потрібні стратегії та механізми для організації простору моделювання. Розділяй і володарюй – це корисний принцип, але залишається відкритим питання, які конкретні стратегії можна застосувати, щоб розділити та структурувати простір моделювання. Існує багато способів моделювання та управління варіативністю, але основні виклики залишаються: інженерам лінійки продуктів необхідно визначити зміни проблемного простору, тобто потреби зацікавлених сторін і бажані функції; варіативність простору рішення, тобто архітектури та компонентів технічного рішення і залежності між цими двома потребами. Незалежно від конкретного підходу моделювання, який використовується, є кілька варіантів структурування та організації простору моделювання:

- Віддзеркалення структури простору рішень. Кожного разу, коли лінійки продуктів моделюються для вже існуючих систем програмного забезпечення, структура доступних ресурсів для повторного використання може стати основою для організації простору моделювання. Можуть

створюватися моделі, що відображають структуру технічного рішення. Це можна зробити шляхом створення окремих моделей мінливості для різних підсистем лінійки продуктів. Наприклад, структура пакету програмної системи або опис існуючої архітектури може служити відправною точкою. Кількість різних моделей має бути невеликою, щоб уникнути негативного впливу на ремонтопридатність і послідовність. Ця стратегія може бути придатною, наприклад, якщо відповідальність розробників і архітекторів для певних підсистем чітко встановлена.

- Розкладання на кілька продуктових ліній. У більшому масштабі складні продукти часто організовуються за допомогою структури з кількома продуктами [43]. Наприклад, можуть існувати окремі продуктові лінії для різних цільових споживачів, наприклад, лінійки мобільних телефонів для людей похилого віку, підлітків і ділових людей. Іншими прикладами є складні системи з інтенсивним використанням програмного забезпечення, такі як автомобілі або промислові підприємства з системою системної архітектури, яка може містити декілька менших ліній продуктів як частину більшої системи. Моделі мають бути визначені для кожної з цих ліній продуктів і підтримуватися узгодженими під час розробки домену та додатків. Ця стратегія часто означає, що різні зацікавлені сторони створюють моделі мінливості для лінійки продуктів, за яку вони відповідають.
- Структурування за типом активів. Інший спосіб роботи з масштабом моделей лінійки продуктів полягає в структуруванні простору моделювання на основі типів активів у домені. Потім можна створювати окремі моделі для різних типів активів лінійки продуктів. Прикладами є моделі варіативності, засновані на варіантах використання, моделі варіативної архітектури або моделі зміненої документації для технічних і призначених для користувача документів. Цей підхід узгоджується з ортогональними підходами, які пропонують використання кількох моделей варіативності, пов’язаних із багатьма моделями атрибутів. Структурування за типом атрибуту дозволяє узгоджено керувати змінами. Однак важливо

враховувати залежності між різними типами артефактів, які можуть спричинити додаткову складність.

- Слідування організаційній структурі. Ця стратегія передбачає дотримання структури організації при створенні моделей лінійки продуктів. У багатьох організаціях архітектурні знання розподіляються між різними зацікавленими сторонами, незалежно від їхніх ролей і обов'язків у процесі розробки. Стверджується, що організації, які розробляють складні системи змушені створювати проекти, які є копіями комунікаційних структур цих організацій. У середовищі кількох команд окремі команди тісно співпрацюють над певними аспектами лінійки продуктів. Таким чином, гарною стратегією може бути структурування простору моделювання лінійки продуктів на основі командної структури, щоб відобразити проблеми моделювання залучених груп зацікавлених сторін. Однак створення моделей лінійки продуктів, керованих зацікавленими сторонами, може легко збільшити надмірність у моделях.
- Розгляд наскрізних проблем. Використання концепцій аспектно-орієнтованої розробки для структурування моделей лінійки продуктів корисно, коли потрібно описати багато наскрізних функцій. Аспектно-орієнтоване моделювання лінійки продуктів можна використовувати для моделювання мінливості простору як проблеми, так і рішення. Наприклад, є відомий підхід, який передбачає створення моделі основних характеристик, спільних для всіх продуктів, і визначення моделей мінливості аспектів для специфічних для продукту характеристик, спільних лише для деяких продуктів. Однак складні аспектні залежності можуть призвести до труднощів керування їх взаємодією.
- Орієнтація на потреби ринку. Структурування простору моделювання також може залежати від бізнесу та менеджменту, наприклад, з точки зору маркетингу [44] або управління продуктом [45]. Зосередження моделювання змін на бізнес-міркуваннях полегшує спілкування з клієнтами. У поєднанні з іншими стратегіями цей підхід може підтримувати спілкування

між клієнтами та продавцями. Якщо слідувати цій стратегії в чистому вигляді, моделі часто не пов’язані з технічним рішенням, що призводить до проблем при спробі зрозуміти фактичну реалізацію змін.

Висновки до розділу 3

Отже, в цьому розділі представлено опис інстументу DecisionKing, що є невід’ємною частиною сервісів моделювання, який було обрано для забезпечення такої інтегрованої підтримки побудови метамоделей. DecisionKing - це мета-інструмент для моделювання змін, що забезпечує середовище моделювання варіативності для сімейства доменів. Налаштувавши даний інструмент можна значно скоротити кількість часу, зусиль і ресурсів, необхідних для розробки та підтримки інструментів моделювання змін. Також запропоновано підхід який базується на простому рішенні: маленьку модель легше обслуговувати, ніж велику. Замість створення однієї великої моделі змін ми використовуємо фрагменти моделі для опису змін вибраних частин системи.

ВИСНОВКИ

В представленій кваліфікаційній роботі розглянуто процеси імплементації підходу на основі моделей для побудови гнучких та адаптивних програмних рішень та сервісів. Проведено дослідження суть якого полягає в комплексному поєднанні підходів до моделювання потреб і вимог стейкхолдерів, характеристик та особливостей програмного продукту, архітектурних елементів та інших ресурсів. Використовуючи моделі варіативності як засобу для отримання неявних знань, необхідних для конфігурації, адаптації та моніторингу систем програмного забезпечення, запропоновано підхід до моделювання домену, який виходить за межі можливостей існуючих інструментів і методів. Це дослідження зосереджене на ключових питаннях розвитку та підтримки ліній продуктів їх даних і моделей. Створено метод моделювання варіативності, який може включати різні рівні абстракції в одну модель, відповідно даний метод є незалежним від практик у різних організаційних умовах. Визначено необхідні можливості для відповідної підтримки інструментів і використано прототипну реалізацію інструментів для підтвердження концепції. Інструменти які застосовуються для дослідження є гнучкими, адаптованими та розширюваними для підтримки різних сценаріїв моделювання та розвитку. За допомогою інструменту VecisionKing виконано об’єднання фрагментів моделі варіативності. Це унікальна функція VecisionKing, яка підтримує децентралізоване та несинхронізоване створення моделей варіативності, структуруючи простір моделювання і забезпечує необхідний простір для цілей багатокомпонентного моделювання. Проведено об’єднання різних фрагментів моделі в одну повну модель варіативності, оскільки фрагменти моделі складаються з невирішених посилань і представляють лише часткову модель змін. Компонент злиття моделі відповідає за напівавтоматичне вирішення невизначеностей, які виникають під час злиття.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Abi-Antoun, M., Aldrich, J., Nahas, N., Schmerl, B., & Garlan, D. 2006. Differencing and Merging of Architectural Views. In: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06). Tokyo,Japan: IEEE Computer Society.

2. Allen, R., & Garlan, D. 1997. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, 6(3), 213–249.

3. Anastasopoulos, M., & Gacek, C. 2001. Implementing product line variabilities. Pages 109–117 of: 2001 Symposium on Software reusability: putting software reuse in context. Toronto, Canada: ACM Press.

4. Anastasopoulos, M., & Muthig, D. 2004. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. Pages 141–156 of: Bosch, J., & Krueger, C. (eds), 8th International Conference on Software Reuse (ICSR 2004), vol. LNCS 3107. Madrid, Spain: Springer Berlin Heidelberg.

5. Asikainen, T., Männistö, T., & Soininen, T. 2006. A Unified Conceptual Foundation for Feature Modelling. Pages 31–40 of: 10th International Software Product Line Conference (SPLC 2006). Baltimore, MD, USA: IEEE Computer Society.

6. Atkinson, C., Bayer, J., & Muthig, D. 2000. Component-based product line development: the KobrA Approach. Pages 289–310

7. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J., & Zettel, J. 2002. Component-Based Product Line Engineering with UML. Addison-Wesley.

8. Basili, V. R. 1993. The experimental paradigm in software engineering. In Experimental Software Engineering Issues: Critical Assessment and Future Directives. In: Dagstuhl-Workshop, H. Dieter Rombach, Victor R.

Basili, and Richard Selby (eds). Lecture Notes in Computer Science:Springer-Verlag 1993.

9. Batory, D. 2005. Feature Models, Grammars, and Propositional Formulas. Pages 7–20 of: 9th International Software Product Line Conference (SPLC 2005), vol. LNCS 3714. Rennes, France:Springer Berlin Heidelberg.

10. Batory, D., Benavides, D., & Ruiz-Cortez, A. 2006. Automated analysis of feature models: challenges ahead. Communications of the ACM, 49(12), 45–47.

11. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., & DeBaud, J. 1999. PuLSE: a methodology to develop software product lines. Pages 122–131 of: SSR ’99: Proceedings of the 1999 symposium on Software reusability. New York, NY, USA: ACM.

12. Benavides, D., Segura, S., Trinidad, P., & Ruiz-Cortes, A. 2005. Using Java CSP Solvers in the Automated Analyses of Feature Models. In: Generative and Transformational Techniques in Software Engineering (GTTSE’05).

13. Bentley, J. 1986. Little Languages. Communications of the ACM, 29(8), 711–721.

14. Berg, K., Bishop, J., & Muthig, D. 2005. Tracing software product line variability: from problem to solution space. Pages 182–191 of: SAICSIT ’05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries. , Republic of South Africa: South African Institute for Computer Scientists and Information Technologists.

15. Bézivin, J., & Gerbé, O. 2001. Towards a Precise Definition of the OMG/MDA Framework. Pages 273–281 of: ASE ’01: Proceedings of the 16th IEEE international conference on Automated software engineering. Washington, DC, USA: IEEE Computer Society.

16. Bosch, J. 2000. Design and use of software architectures: adopting and evolving a productline approach. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

17. Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J. H., & Pohl, K. 2002. Variability Issues in Software Product Lines. Pages 13–21 of: PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering. London, UK: Springer-Verlag.

18. Broek, Pim van den, Galvao, Ismenia, & Noppen, Joost. 2008. Elimination of Constraints from Feature Trees. In: First Workshop on Analyses of Software Product Lines in conjunction with Software Product Line Conference, SPLC 2008.

19. Burgstaller, B. 2008. Runtime Adaptation of Service-oriented Systems Based on Product Line Variability Models. Institute for Systems Engineering and Automation, vol. Masters thesis. Linz: Johannes Kepler University.

20. Campbell, G. H., Faulk, S. R., & Weiss, D. M. 1990. Introduction To Synthesis. Tech. rept. Software Productivity Consortium, Herndon, VA, USA.

21. Cechticky, V., Pasetti, A., Rohlik, O., & Schaufelberger, W. 2004. XML-Based Feature Modelling. Pages 101–114 of: Bosch, J., & Krueger, C. (eds), 8th International Conference on Software Reuse (ICSR 2004), vol. LNCS 3107. Madrid, Spain: Springer Berlin Heidelberg.

22. Charles, P., Fuhrer, R. M., & Sutton, S. M. 2007. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. Pages 485–488 of: ASE '07: Proceedings of the twentysecond IEEE/ACM Int'l Conf. on Automated software engineering. New York, NY, USA: ACM.

23. Chen, P. P. 1976. The entity-relationship model—toward a unified view of data. ACM Transactions on Database Systems (TODS), 1(1), 9–36.

24. Clayberg, E., & Rubel, D. 2006. Eclipse: Building Commercial-Quality Plugins. 2 edn. The Eclipse Series. Addison-Wesley Professional.

25. Clements, P., & Northrop, L. 2001. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering, Addison-Wesley.

26. Clotet, R., Franch, X., López, L., Marco, J., Seyff, N., & Grünbacher, P. 2007a. The Meaning of Inheritance in i*. In: 17th International Workshop on Agent-oriented Information Systems (AOIS-2007).

27. Clotet, R., Xavier, F., Grünbacher, P., López, L., Marco, J., Quintus, M., & Seyff, N. 2007b. Requirements Modelling for Multi-Stakeholder Distributed Systems: Challenges and Techniques. In: RCIS'07: 1st IEEE Int. Conf. on Research Challenges in Information Science.

28. Clotet, R., Dhungana, D., Franch, X., Grünbacher, P, López, L., Marco, J., & Seyff, N. 2008. Dealing with Changes in Service-Oriented Computing Through Integrated Goal and Variability Modeling. Pages 43–52, http://www.icb.uni–due.de/researchreports/ of: Second International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2008). Essen, Germany: ICB-Research Report No. 22, University of Duisburg Essen.

29. Consortium, Software Productivity. 1991. Synthesis Guidebook. Tech. rept. SPC- 91122-MC. Herndon, Virginia: Software Productivity Consortium.

30. Conway, M.E. 1968. How Do Committees invent? Datamation, 14(4), 28–31.

31. Coplien, J., Hoffman, D., & Weiss, D. 1998. Commonality and Variability in Software Engineering. IEEE Software, 15(6), 37–45.

32. Czarnecki, K., & Antkiewicz, M. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. Pages 422–437 of: Proceedings of the Fourth International Conference on Generative Programming and Component Engineering. Tallinn, Estonia: Springer-Verlag, LNCS 3676.

33. Czarnecki, K., & Eisenecker, U.W. 2000. Generative Programming: Methods, Techniques, and Applications. Addison-Wesley.

34. Czarnecki, K., & Kim, C.H.P. 2005. Cardinality-Based Feature Modeling and Constraints: A Progress Report. Pages 1–9 of: International Workshop on Software Factories at OOPSLA' 05. San Diego, USA: ACM Press.

35. Czarnecki, K., & Pietroszek, K. 2006. Verifying feature-based model templates against well-formedness OCL constraints. Pages 211–220 of: GPCE '06:

Proceedings of the 5th international conference on Generative programming and component engineering. New York, NY, USA: ACM.

36. Czarnecki, K., Helson, S., & Eisenecker, U.W. 2004. Staged configuration using feature models. Pages 266–283 of: Nord, R. (ed), Lecture Notes in Computer Science, Software Product Lines, Third International Conference (SPLC 2004), vol. LNCS 3154. Springer-Verlag.

37. Czarnecki, K., Helsen, S., & Eisenecker, U. 2005. Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice, 10(1), 7–29.

38. Czarnecki, K., Kim, C. H. P., & Kalleberg, K. T. 2006. Feature Models are Views on Ontologies. Pages 41–51 of: SPLC ’06: Proceedings of the 10th International on Software Product Line Conference. Washington, DC, USA: IEEE Computer Society.

39. Dashofy, E., Asuncion, H., Hendrickson, S., Suryanarayana, G., Georgas, J., & Taylor, R. 2007. ArchStudio 4: An Architecture-Based Meta-Modeling Environment. 29th International Conference on Software Engineering. ICSE’07 Companion., May, 67–68.

40. Dashofy, E.M., van der Hoek, A., & Taylor, R.N. 2001. A Highly-Extensible, XML-Based Architecture Description Language. Pages 103–112 of: Working IEEE/IFIP Conference on Software Architecture (WICSA’01). Amsterdam, The Netherlands: IEEE Computer Society.

41. Dashofy, E.M., van der Hoek, A., & Taylor, R.N. 2002. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In: International Conference on Software Engineering (ICSE 2002).

42. Dhungana, D., Rabiser, R., Grünbacher, P., Prähofer, H., Federspiel, C., & Lehner, K. 2006. Architectural Knowledge in Product Line Engineering: An Industrial Case Study. Pages 186–197 of: 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Cavtat/Dubrovnik, Croatia: IEEE CS.

43. Dhungana, D., Rabiser, R., & Grünbacher, P. 2007a. Decision-Oriented Modeling of Product Line Architectures. In: Sixth Working IEEE/IFIP Conference on Software Architecture.

44. Dhungana, D., Grünbacher, P., & Rabiser, R. 2007b. Domain-specific Adaptations of Product Line Variability Modeling. In: IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences.

45. Dhungana, D., Rabiser, R., Grünbacher, P., Lehner, K., & Federspiel, C. 2007c.

46. DOPLER: An Adaptable Tool Suite for Product Line Engineering. Pages 151–152 of: 11th International Software Product Line Conference (SPLC 2007), Tool Demonstration, vol. Second Volume. Kyoto, Japan: Kindai Kagaku Sha Co. Ltd.

47. Dhungana, D., Rabiser, R., Grünbacher, P., & Neumayer, T. 2007d. Integrated Tool Support for Software Product Line Engineering. In: Tool Demonstration, 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007).

48. Dhungana, D., Neumayer, T., Grünbacher, P., & Rabiser, R. 2008. Supporting Evolution of Product Line Architectures With Variability Model Fragments. In: Working IEEE/IFIP Conference on Software Architecture, WICSA 2008.

метадані

Заголовок

Імплементація підходу на основі моделей для побудови гнучких та адаптивних програмних рішень та сервісів

Автор

Скірчук В.В. Науковий керівник / Експерт

підрозділ

King Danylo University

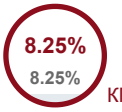
Тривога

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

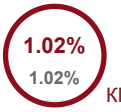
Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	82

Обсяг знайдених подібностей

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



КП 1



КП 2

25
Довжина фрази для коефіцієнта подібності 2

15636
Кількість слів

121294
Кількість символів

Подібності за списком джерел

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

10 найдовших фраз Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	http://repository.ukd.edu.ua/bitstream/handle/123456789/388/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0%20%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0%20%D0%9B%D0%B8%D1%82%D0%B2%D0%B0%D0%BA.pdf?sequence=1	45	0.29 %
2	http://repository.ukd.edu.ua/bitstream/handle/123456789/388/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0%20%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0%20%D0%9B%D0%B8%D1%82%D0%B2%D0%B0%D0%BA.pdf?sequence=1	35	0.22 %
3	http://link.springer.com/chapter/10.1007%2F978-3-642-04211-9_21	29	0.19 %
4	https://www.fh-krems.ac.at/fachhochschule/team/deepak-dhungana/	26	0.17 %