КВАЛІФІКАЦІЙНА РОБОТА

Група МІПЗс-22

Білоус В.В.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Білоус Василь Васильович

УДК 004.4

Порівняльний аналіз моделей та методів інтеграції та агрегації процесів розробки програмного забезпечення

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль

_____Стисло О.В.

(підпис, дата, розшифрування підпису)

Студент

_____ Білоус 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 2023 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Білоусу Василю Васильовичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Порівняльний аналіз моделей та методів інтеграції та агрегації процесів розробки програмного забезпечення

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. Огляд та аналіз фреймворків агрегації процесів розробки
2. Класифікація та структуризація програмних методів в контексті задач проектування
3. Розробка інтеграційних моделей процесів розробки ПЗ
4. Інтеграція методів розробки ПЗ із застосуванням правил

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 фреймворків агрегації процесів розробки	26.09.2023	Виконано
2.	Класифікація та структуризація програмних методів в контексті задач проектування	20.10.2023	Виконано
3.	Розробка інтеграційних моделей процесів розробки ПЗ	15.11.2023	Виконано
4.	Інтеграція методів розробки ПЗ із застосуванням правил	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 _____ Білоус 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
28	Загальне середовище мета-CASE на основі методів	59	Централізована та децентралізована архітектури
34	Методика інтеграції знань програмної інженерії	61	Інструментальна інтеграція як випадок інтеграції методу
39	Зв'язок через «програмну шину» ESF	64	Можливості інтеграції ViewPoints framework
39	Централізована та децентралізована інтеграційні архітектури	66	Учасники розробки програмного забезпечення
48	Шари / слоти ViewPoint	67	Розробка методу з використанням концепції ViewPoint

АНОТАЦІЯ

Кваліфікаційна робота присвячена дослідженню моделей та виконання порівняльного аналізу моделей та методів інтеграції та агрегації процесів розробки програмного забезпечення шляхом розробки підходу до вимірювання та контролю процесів розробки, виявлення низькоякісного програмного забезпечення та формування методології збору загальних, істотних ознак.

В першому розділі виконано огляд та аналіз методів та інструментів агрегації процесів розробки програмного забезпечення. Виконано дослідження та класифікацію методів розробки програмного забезпечення, проведено системний аналіз інструментів та фреймворків агрегації для розробки програмного забезпечення.

В другому розділі проведена класифікація та структуризація програмних методів в контексті задач розробки програмного забезпечення, досліджені концепції інтеграції та агрегації в процесах розробки програмного забезпечення. Описані фреймворки, архітектура та механізми інтеграції процесів розробки, виконано дослідження концепції побудови ViewPoints Framework, наведені характеристики, інфраструктура, шаблони та методи побудови ViewPoints Framework.

В третьому розділі здійснена реалізація моделей інтеграції та агрегації процесів розробки програмного забезпечення, виконано представлення архітектури та моделі об’єкта в інтеграційному фреймворці розробки. Здійснено розробку методу на основі ViewPoints та проведена імплементація методів із застосуванням правил.

КЛЮЧОВІ СЛОВА: ІНТЕГРАЦІЯ ПРОЦЕСІВ, ОБ’ЄДНАННЯ ПРАВИЛ, ДІАГРАМА ДАНИХ, АГРЕГАЦІЯ ПРОЦЕСІВ, МЕТОДОЛОГІЯ, АРХІТЕКТУРА, КЛАСИФІКАЦІЯ, ШАБЛОН ПРОЕКТУВАННЯ.

SUMMARY

The qualification work is dedicated to researching models and performing a comparative analysis of models and methods of integration and aggregation of software development processes by developing an approach to measuring and controlling development processes, identifying low-quality software, and forming a methodology for collecting common, essential and desirable features.

In the first section, an overview and analysis of methods and tools for the aggregation of software development processes is carried out. A study and classification of software development methods was carried out, a system analysis of aggregation tools and frameworks for software development was carried out.

In the second chapter, the classification and structuring of software methods in the context of software development tasks is carried out, the concepts of integration and aggregation in software development processes are studied. The frameworks, architecture and mechanisms of integration of development processes are described, the concept of building ViewPoints Framework is studied, the characteristics, infrastructure, templates and methods of building ViewPoints Framework are given.

In the third section, the integration and aggregation models of software development processes were implemented, the architecture and object model in the development integration framework were presented. The method based on ViewPoints was developed and the methods using rules were implemented.

KEY WORDS: PROCESS INTEGRATION, COMBINATION OF RULES, DATA DIAGRAM, PROCESS AGGREGATION, METHODOLOGY, ARCHITECTURE, CLASSIFICATION, DESIGN PATTERN.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AI - Artificial Intelligence

APSE - Ada Programming Support Environment

CAIS - Common APSE Interface Set

CAME - Computer-Aided Method Engineering

CASE - Computer-Aided Software Engineering

CDA - Constructive Design Approach

CDIF - CASE Data-Interchange Format

CMM - Capability Maturity Model

CORE - Controlled Requirements Expression

CSCW - Computer-Supported Cooperative Work

CSP - Communicating Sequential Processes

DAI - Distributed Artificial Intelligence

DBMS - Database Management System

DFD - Data Flow Diagram

DRL - Decision Representation Language

ER - Entity-Relationship

ESF - Eureka Software Factory

ESPRIT - European Strategic Programme for Research and Development in Information Technology

I-CASE - Integrated CASE

IPSE - Integrated Project Support Environment

JSD - Jackson System Design

MLP - Mixed Language Programming

OMS - Object Management System

OOA - Object-oriented Analysis

OOD - Object-Oriented Design

OOP - Object-Oriented Programming

PCTE - Portable Common Tool Environment

PTI - Public Tool Interface

RCS - Revision Control System

RML - Requirements Modelling Language

RPC - Remote Procedure Call

SADT - Structured Analysis and Design Technique

SCCS - Source Code Control System

SCS - Structured Common Sense

SD - Structured Design

SEI - Software Engineering Institute

SLI - Specification Level Interoperability

SREM - Software Requirements Engineering Methodology

SSADM - Structured Systems Analysis and Design Methodology

TBK - Tool Builder’s Kit

UIMS - User-Interface Management System

UTS - Universal Type System

VOA - Viewpoint-Oriented Analysis

VOSE - ViewPoint-Oriented Software/Systems Engineering

VSF - Virtual Software Factory

YACC - Yet Another Compiler-Compiler

ВСТУП

Актуальність теми дослідження. У зв’язку з так званою «кризою програмного забезпечення», яка є болючою реальністю для багатьох фірм у комп’ютерній індустрії, навряд чи існувала більш нагальна потреба у виробництві високоякісного програмного забезпечення до встановленого терміну. З цією метою вчені в сфері ІТ надали набір потужних комп’ютерних мов і технік, що полегшує розробку модульного, зручного для обслуговування та ефективного коду. З іншого боку, розробники програмного забезпечення багатьма способами намагалися наслідувати своїх апаратних колег, забезпечуючи формалізм, якого так часто бракує великим проектам розробки програмного забезпечення. Із постійно зростаючим проникненням складних комп’ютерних систем у повсякденне функціонування суспільства розробка програмного забезпечення стала важливою галуззю.

Складні системи, як правило, розгортають кілька технологій і потребують кількох учасників для їх розробки. Таким чином, хоча для розробки таких систем необхідні систематичні та структуровані підходи, необхідно також підтримувати неминуче поширення різних поглядів і точок зору на проблеми та області вирішення. Ці погляди незмінно виражаються та розробляються з використанням різних нотацій та стратегій розвитку відповідно.

В даній роботі розглядається питання управлінню складністю, поділу проблем та інтегрованому, систематичному розробці програмного забезпечення. В роботі описано складність розробки програмного забезпечення у вигляді «проблеми багатьох точок зору», і пропонуються межі, в яких цю проблему можна вирішити, а також спроби узгодити бажаний поділ проблем, передбачений структурою з інтеграцією, необхідною для систематичної розробки програмного забезпечення. Таке узгодження досягається шляхом використання кількох точок зору під час розробки,

забезпечуючи при цьому інтеграцію методів, за допомогою яких ці точки зору розробляються.

В даний час широкого поширення набула об'єктно-орієнтована адаптація метрики розробки в численних областях застосування, оскільки теоретично можна довести, що показники є дійсними в тому сенсі, що вони точно вимірюють атрибути програмного забезпечення, для якого вони розроблені, а також можуть бути перевірені емпірично. Існують думки дослідників, що «метрики дизайну можуть охоплювати всі фактори якості» і це може допомогти вирішити багато актуальних питань.

Щоб звести до мінімуму використання ресурсів та підвищення ефективності витрат, мінімальний набір показників є загальною ознакою проектних метрик. Властиве використання релевантності і значення метрик хорошого дизайну для тієї ж системи змінюватиметься не залежно від часу та людей, а від основної зміни парадигми у сфері розробки програмного забезпечення. Це також очевидно, що метрики повинні зберігати всі інтуїтивні уявлення про атрибути і спосіб розрізнення показників між сутностями. Тому виходить набір істотних і бажаних особливостей, які можуть бути ідентифіковані а, отже, забезпечені метрикою розробників. Можна перерахувати абстракції, як основні характеристики для проектування якості метрики: здатність охоплювати всі аспекти та характеристики конструкції проектованого рішення.

Показники програмного забезпечення є невід’ємною частиною практики в розробці програмного забезпечення. Розроблені показники із задокументованими цілями можуть допомогти в отриманні необхідної інформації, продовження вдосконалення програмних продуктів, процесів та послуг, зосереджуючись на тому, що для цього важливо замовнику. Таким чином, проектування є найважливішим і критичним крок до розвитку бажаних показників якості дизайну. Теоретична перевірка метрик програмного забезпечення надає підтверджуючі докази того, чи така міра дійсно фіксує внутрішні атрибути, що вимірюються. Головна мета в перевірці

полягає в тому, щоб оцінити, чи метрика фактично вимірює те, що має на меті виміряти. У контексті емпіричного дослідження, теоретична перевірка метрик встановлює їх конструктивну валідність, тобто це «доводить», що вони є дійсними мірами для конструкції, які використовуються як змінні в дослідженні.

Мета і задачі дослідження. Проектування конструктів розробки, ефективне використання конструктів-шаблонів для зниження архітектурної складності проекту, специфічний дизайн програми та тести покращення через структуру, що можуть бути підняті на час проектування показників якості, щодо здатності представляти різні аспекти вимірюваної системи.

Метою роботи є виконання порівняльного аналізу моделей та методів інтеграції та агрегації процесів розробки програмного забезпечення і розробка нової концепції побудови програмних рішень на основі фреймворку ViewPoints, суть якого полягає в тому, щоб відійти від централізованої архітектури до децентралізованої.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

- Виконати огляд та системний аналіз фреймворків агрегації процесів розробки;
- Провести дослідження концепції побудови ViewPoints Framework;
- Побудувати класифікацію та структуризацію програмних методів в контексті задач проектування та розробки;
- Дослідити характеристики, інфраструктуру, шаблони та методи побудови ViewPoints Framework;
- Реалізувати проектування методу розробки програмного забезпечення на основі концепції ViewPoint;
- Виконати розробку інтеграційних моделей процесів розробки програмного забезпечення.

Об’єктом дослідження є самі моделі та методи інтеграції та агрегації процесів розробки програмного забезпечення.

Предметом дослідження є моделі, методи, підходи інтеграції та агрегації процесів розробки програмного забезпечення з використанням інтеграційного фреймворку ViewPoints.

Методи дослідження. В магістерській роботі використані методи якісної інтерпретації, обслуговування всіх аспектів об'єктно-орієнтованого проектування, охоплення всіх факторів якості, та методи оптимізації процесів оцінки якості.

Наукова новизна одержаних результатів полягає у тому, що на основі порівняльного аналізу інтеграційних методів розробки програмного забезпечення розроблено метод ідентифікації та визначення предметних областей за допомогою правил засобами фреймворку ViewPoints які розподіляються між різними шаблонами, що призводить до повністю розповсюджуваної конфігурації ViewPoints, яка є ефективною для розробки складних систем.

Практичне значення одержаних результатів полягає в розробці структури ViewPoints для розподіленої багатонапрямленої розробки програмного забезпечення, що складається з структурних елементів фреймворку, ViewPoints і шаблонів ViewPoint, з можливістю підтримки ViewPoint-Oriented Software Engineering (VOSE).

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 “Методи тестування web-додатків”.

Структура. Кількість розділів – 3. Загальний обсяг основної частини – 86 сторінок. Список використаних джерел містить – 52 позиції.

РОЗДІЛ 1. ОГЛЯД ТА АНАЛІЗ МЕТОДІВ ТА ІНСТРУМЕНТІВ АГРЕГАЦІЇ ПРОЦЕСІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Аналіз предметної області розробки програмних систем та сервісів

Великі та складні програмні комплекси зазвичай розгортають кілька з використанням декількох технологій із залученням для їх розробки великої кількості команд і розробників. Як і у випадку будь-якої великої розробки, розробка складної системи має бути систематичною та структурованою, щоб керувати цією складністю, а також для того, щоб полегшити майбутнє обслуговування та еволюцію системи [2].

Метод інженерії програмного забезпечення передбачає дії з розробки та евристики для розгортання однієї або кількох нотацій для специфікації складної системи [3]. Однак метод розробки програмного забезпечення — це більше, ніж просто набір нотацій і відповідних процедур для їхнього розгортання. В [4] пропонують принаймні три типи властивостей, які можна використовувати для характеристики та оцінки методу розробки програмного забезпечення. Технічні властивості стосуються тих аспектів методу, які стосуються його філософії, охоплення життєвого циклу, структури та особливостей (таких як позначення, процедури, заходи та рекомендації). Властивості використання відносяться до тих аспектів методу, які виражають його застосовність в організації та його придатність для конкретної мети. До них відноситься наявність автоматизованої підтримки інструментів, а також управлінські властивості пов’язані з підтримкою методу для практик управління розробкою програмного забезпечення, таких як оцінка витрат, планування проекту та персонал.

Таким чином, очевидно, що метод розробки програмного забезпечення є великим і складним об’єктом і його придатність для будь-якої конкретної

мети дуже сильно залежить від численних загальних і предметно-специфічних характеристик [5]. Дослідження методів, наприклад їх характеристика або каталогізація, можна назвати «методологічною інженерією» і відрізняти її від «методологічної інженерії», яка пов’язана з проектуванням і створенням конкретних методів. Часто використовують термін «моделювання методів» у представленні структури для моделювання методів розробки програмного забезпечення та розробляють формальну мову для опису таких методів [6]. Така формалізація методу розробки програмного забезпечення може мати такі корисні застосування, як автоматизована перевірка, запис обґрунтувань проекту та повторне використання.

Проте, як правило, недостатньо прийняти метод розробки програмного забезпечення, щоб успішно розробити велику та складну систему. Для ефективного використання методу він повинен підтримуватися комп’ютерними інструментами, які допомагають автоматизувати та підтримувати різні дії з розробки програмного забезпечення, передбачені цим методом. Інструменти автоматизованої розробки програмного забезпечення (CASE) традиційно забезпечували автоматизовану підтримку методів у формі редакторів позначень, засобів перевірки узгодженості, генераторів коду тощо. Однак вони не змогли поєднати цей вид підтримки з автоматизованою підтримкою процесу розробки програмного забезпечення, зокрема, інструменти CASE не можуть надати розробникам програмного забезпечення «керівництво методами» [7] тобто вказівки щодо того, коли та за яких обставин доцільно виконувати будь-які кроки методу. Крім того, ті інструменти, які мали модель процесу розробки програмного забезпечення, вбудовану в них, було важко адаптувати до індивідуальних які застосовують в багатьох організаціях. Щоб зробити їх більш «гнучкими», багато комерційних CASE-інструментів було зведено до інструментів з деякою перевіркою синтаксичної узгодженості, вбудованою в них [8].

Складність гетерогенних, складних систем, тобто систем, що використовують багато технологій і учасників розробки вимагає декомпозиції

як проблемних областей так і областей вирішення. З одного боку, різні учасники розробки можуть звертатися до різних аспектів системи, що розробляється. Крім того, погляди та точки зору різних учасників щодо областей перетину або збігу також можуть відрізнятися. З іншого боку, саме існування різних учасників часто означає, що вони висловлюють свої проблеми різними мовами та використовують різні стратегії для цього. Це ускладнює проблему перевірки узгодженості між різними поглядами чи точками зору учасників. Хоча може бути прийнятий підхід, що враховує неузгодженості і численні точки зору (і прийнятий багатьма організаціями, які бажають запровадити політику розвитку) але він є нереалістичним, а тому розробники програмного забезпечення віддають перевагу працювати з кількома технологіями та мовами у яких допускається непослідовність [9]. Насправді, навіть у світі баз даних, де традиційно наголос робиться на побудові універсальних метамоделей або схем для бази даних, потреба в кількох представленнях і схемах завжди існує, а гетерогенні мультибази даних були досліджені [10], то були запропоновані нові методи інтеграції схем і представлень [11, 12].

Насправді, більше не реально розглядати розробку програмного забезпечення як централізований послідовний набір дій, а найчастіше це спільне зусилля багатьох учасників розробки, які одночасно працюють у розподіленому середовищі. Таким чином, структура, в якій відбувається така розробка, повинна бути здатна вирішувати конкретні проблеми кооперативної роботи з комп’ютерною підтримкою (CSCW), розподілених систем [13] і паралелізму [14].

Описаний вище сценарій, у якому кілька учасників мають різні погляди на програмну систему, яку вони розробляють, можна назвати «багатонапрямленою розробкою програмного забезпечення». Методи розробки програмного забезпечення, призначені для підтримки такої багатонапрямленої розробки, повинні бути здатні обробляти інформацію від великої кількості учасників, їх поглядів, стратегій розробки та нотацій таким

чином, що результуючі фрагменти специфікації є слабко пов’язаними, але когерентними. Цього можна досягти шляхом забезпечення інтеграції одного або кількох методів, які використовуються для розробки таких багатоаспектних специфікацій або систем. Отже, необхідна ефективна інтеграція методів для багатонапрямленої розробки програмного забезпечення. Крім того, ефективна інтеграція методів повинна підтримуватися ефективною інтеграцією інструментів.

Багатонапрямлена розробка програмного забезпечення створює особливі умови для розробника інструментів. З одного боку, окремі розробники повинні вільно вибирати інструменти, які найбільше підходять для їхніх конкретних цілей, а з іншого боку, ці інструменти повинні мати можливість ефективно спілкуватися та обмінюватися інформацією. Це може вимагати певної форми стандартизації форматів інформації, якою обмінюються і протоколів зв’язку між будь-якими учасниками розробки, що спілкуються (або інструментами, які вони розгортають). У термінології інструментів CASE потрібно визначити зв’язки узгодженості між частковими специфікаціями та, якщо це доцільно, ці зв’язки потрібно перевірити. Необхідно також вирішити питання про те, як діяти за наявності неузгодженості. Типовий CASE-інструмент у таких налаштуваннях може позначати помилку та чекати вирішення конфлікту одним або декількома учасниками розробки. Це може бути невідповідним підходом, якщо вирішення конфліктів не є очевидним.

1.2 Дослідження та класифікація методів розробки програмного забезпечення

Інженерія програмного забезпечення — це дисципліна розробки програмного забезпечення, що включає в себе розробку на ранніх етапах реалізації вимог і специфікацій аж до проектування, впровадження та постійного обслуговування. Інженерія програмного забезпечення забезпечує

та поєднує методи розробки програмного забезпечення, які, у свою чергу, забезпечують методи специфікації, впровадження, забезпечення якості, координації, управління та контролю розробки, а також ефективну автоматизовану підтримку для багатьох дій у цьому процесі. Модель, яка представляє такий процес розробки (його складові дії та їх порядок), зазвичай називається моделлю життєвого циклу розробки програмного забезпечення.

Складність сучасних програмних систем вимагає використання чітко визначених системних підходів до їх розробки.

Метод розробки програмного забезпечення (далі метод) визначає набір процедур і вказівок для специфікації та розробки системи програмного забезпечення. Метод може стосуватися всього або частини життєвого циклу розробки програмного забезпечення. Таким чином, існують методи розробки вимог, такі як методологія розробки вимог до програмного забезпечення (SREM) [11], методи проектування, такі як Structured Design (SD) [12] і методи розробки програмного забезпечення, які включають як аналіз, так і етапи проектування, такі як Structured Systems Analysis and Design Method (SSADM) [13]. Термін методологія іноді використовується для опису набору або системи методів однак у більшості літератури з програмної інженерії терміни «метод» і «методологія» використовуються як синоніми.

Метод також керує розробкою програмного забезпечення, приписуючи, як розгортати одну або більше нотацій. Ці позначення необхідні для вираження системних вимог, проектування та реалізації. Вони також використовуються для опису різноманітних продуктів і документів, створених проектом розробки програмного забезпечення. Фактично, поняття методу нерозривно пов’язане з поняттями нотацій, які є основним інструментом для вираження ідей про систему та її властивостей [14]. Більшість комерційних методів розробки програмного забезпечення використовують принаймні одну нотацію для опису продуктів, які є результатом застосування процедур і умов методу. Метод, який не має однієї

або кількох нотацій в своїй структурі, більш точно називається моделлю життєвого циклу розробки програмного забезпечення.

Протягом останніх років розробки програмного забезпечення було розроблено низку методів і вони використовували різні позначення та процедури для специфікації та розробки програмних систем.

Техніка структурованого аналізу та проектування (SADT) [15] поєднує високографічну мову з систематичним і структурованим способом специфікації систем, що використовують цю мову. Мова має понад 40 функцій, що вимагає розкладання специфікацій на менші одиниці, які легше зрозуміти. Структурований аналіз (SA) [16] — загальна теорія та методи на яких базується SADT — припускає, що «все, важливе повинно бути виражено шістьма або меншою кількістю частин» [16]. Застосування цієї пропозиції в SADT призводить до ієрархічної декомпозиції системи та її контексту зверху вниз. Модель у SA — це набір взаємопов’язаних діаграм, і підтримується кілька моделей, а кожна модель має орієнтацію, яка включає контекст, точку зору та мету.

Метод контрольованих вимог (CORE) [17] використовує чотири різні позначення діаграм і структурований текст на семи окремих етапах методу, щоб визначити та проаналізувати системні вимоги. Перший етап (визначення проблеми) визначає повноваження клієнта, бізнес-цілі, поточні проблеми системи, нові функції системи, майбутнє розширення, витрати та часові масштаби. Вони описані текстово в структурованій формі.

Наступний етап (структурування точки зору) ідентифікує та організовує сутності обробки інформації у проекті. Ці точки зору впорядковано в ієрархію точок зору за допомогою простого графічного позначення. Кожна точка зору потім аналізується на наступному етапі (табличний збір), щоб визначити джерела вхідних даних і призначення виходів до та від кожної дії, що виконується кожною точкою зору. Ця інформація відображається в серії табличних діаграм - по одній для кожної визначеної точки зору. Потім на наступному етапі (структурування даних) будується діаграма структури

даних для кожної точки зору, на якій дані (вхідні та вихідні дані на попередньому етапі) упорядковуються. Діаграми структури даних – це знову ж таки прості графічні ієрархічні представлення цієї інформації. Наступні два етапи (моделювання однієї та комбінованої точки зору) об'єднують інформацію, виражену на попередніх двох етапах.

Усі графічні етапи CORE мають структуровані текстові анотації, пов'язані з ними і розроблені за допомогою рекомендованих інструкцій та евристик. Хоча етапи представлені послідовно, аналіз вимог за допомогою CORE є ітеративним, як правило, одночасним процесом.

Конструктивний підхід до проектування (CDA) [18] для розробки розподілених систем є методом, структурованим відповідно до процедурних, а не нотаційних алгоритмів. Метою методу є управління розробкою розподіленої системи шляхом відокремлення структури системи, як набору компонентів та їх взаємозв'язків, від функціональних описів окремих компонентів. Розробники стверджують, що це полегшує опис, побудову та еволюцію розподілених систем.

Метод передбачає п'ять рекурсивних неформальних кроків, які переводять розробника системи від високорівневого архітектурного проекту до працюючої розподіленої системи. Цей метод менше залежить від певної нотації, ніж більшість методів, оскільки початковий дизайн може бути викладений дуже вільно в термінах «компонентів» та їхніх основних потоків даних. Управління (синхронізація) між компонентами вводиться на етапі специфікації інтерфейсу, а за ним слідує етап розробки компонентів, на якому ідентифікуються типи компонентів і функціональна поведінка «примітивних» компонентів описується мовою вибору. Конфігурація системи потім створюється шляхом інстанціювання та взаємозв'язку компонентів для формування розподільних логічних вузлів. Знову ж таки, така конфігурація може бути описана мовою конфігурації за вибором розробника. Нарешті, модифікація та еволюція системи здійснюється шляхом заміни або додавання вузлів.

Дизайн системи Джексона (JSD) [19] поєднує чіткі описові нотації з процедурами щодо того, як їх використовувати. Метод поєднує об'єктно-орієнтований підхід до проектування з функціональною декомпозицією та намагається розглянути більшість аспектів життєвого циклу розробки програмного забезпечення – від аналізу до впровадження. JSD складається з трьох окремих етапів: етап моделювання, мережений етап та етап впровадження. На етапі моделювання будуються графічні (деревоподібні) діаграми структури процесу, в яких аналізується процес представлений у термінах дій (або подій), які можуть впливати на об'єкти. Під час мережевої стадії будується (графічна) мережа специфікації системи, в якій уся система описується як мережа взаємопов'язаних комунікуючих процесів, а на етапі реалізації мережа процесів перетворюється на послідовну реалізацію мовою програмування за вибором розробників.

В таблиці 1.1 наведено чотири методи, описані вище та таблиці використання нотацій і процедур, а також міру, в якій надано рекомендації чи евристики.

Таблиця 1.1

Відмінності в чотирьох методах розробки програмного забезпечення

Method	Notations	Procedures	Guidelines
SADT	Actigrams	Few fixed procedures ("a discipline of thought" [Ross 1985]).	Many heuristics.
CORE	1. Structured text 2. Viewpoint structuring 3. Tabular collection 4. Data structuring 5. Single viewpoint modelling 6. Combined viewpoint modelling 7. Structured text	Documented step-by-step procedures for constructing specifications using notations provided.	Many heuristics, guidelines and examples, but there are ambiguities in the notations and the procedures for deploying them.
CDA	1. Configuration structure 2. Configuration language 3. Programming language(s)	1. Structure & component identification 2. Interface specification 3. Component elaboration 4. Construction 5. Modification and evolution	Few guidelines and heuristics, due to relative immaturity of the method.
JSD	1. Process structure diagrams 2. System specification network 3. Programming language	1. Entity-Action identification 2. Network modelling 3. Transformation	Few heuristics, but many examples in the literature and industry.

Таблиця 1.1 призначена не як конкурентне порівняння методів, а як ілюстрація різного акценту, який ці різні методи приділяють трьом критеріям класифікації, представленим у таблиці. Ці критерії в кінцевому підсумку впливають на структуру методів, спосіб їх розгортання та програми, для яких вони використовуються.

Успішне розгортання методу значною мірою залежить від його здатності керувати розробником програмного забезпечення при розробці системи. Таким чином, «хороший» метод має бути достатньо директивним, щоб можна було рекомендувати, яку діяльність з розробки робити далі, але достатньо гнучким, щоб дозволити розробнику час від часу відхилятися від рекомендованих методичних процедур і вказівок. В ідеалі метод також повинен допомагати розробникам опрацьовувати помилки допущені під час розробки.

1.3 Системний аналіз інструментів та фреймворків агрегації для розробки програмного забезпечення

Використання методу для розробки великої складної системи може бути тривалим і виснажливим процесом. Тому підтримка інструментів є важливою для того, щоб методи були ефективними для великомасштабної розробки програмного забезпечення. Розглянемо роль автоматизованої підтримки інструментів і види інструментів, які доступні для розробки програмного забезпечення на основі методів. Розглянуто три категорії інструментів підтримки:

- інструменти автоматизованої розробки програмного забезпечення (CASE);

- інтегровані середовища підтримки проектів (IPSE);

- інструменти розробки методів (meta-CASE).

У цьому контексті також розглядається роль технології баз даних і системи управління базами даних (СУБД).

Інструменти автоматизованої розробки програмного забезпечення (CASE)

Термін автоматизована програмна інженерія зазвичай використовується для опису широкого діапазону комп'ютерних засобів розробки програмного забезпечення. До них належать прості, автономні інструменти, такі як редактори та відладчики або більш складні інтегровані «середовища», які поєднують декілька інструментів та інфраструктуру (наприклад, служби операційної системи та СУБД) для їх підтримки [20].

За останній час спостерігається невпинне зростання популярності інструментів CASE [21]. Велика кількість таких інструментів була створена для підтримки широкого діапазону діяльності з розробки програмного забезпечення і значний досвід був накопичений у їх використанні. В літературі з програмної інженерії є багато описів щодо прогресу у технології інструментів CASE [21, 22].

Простий інструмент CASE зазвичай підтримує або частково автоматизує діяльність з розробки програмного забезпечення, таку як специфікація з використанням певної нотації. Було визначено відмінність між інструментами верхнього CASE-рівня, які підтримують «початкові» дії розробки, такі як планування, розробка вимог і проектування та інструментами нижнього CASE, які забезпечують підтримку «бек-енд» діяльності розробки таких як впровадження та підтримка. Інструменти нижнього рівня, такі як інтерпретатори та статичні аналізатори, існують уже багато років, а тому вони краще зрозумілі більше використовуються. З іншого боку, інструменти верхнього рівня підтримують такі дії, як розробка вимог, які за своєю природою мають справу з артефактами, які можуть бути нечіткими. Часто під час специфікації зовнішнього коду та проектування застосовуються високографічні нотації, які, менш формальні та менш піддаються автоматизованій підтримці. Таким чином, інструменти верхнього рівня традиційно надають базові (діаграмні) редактори та засоби перевірки узгодженості для перевірки синтаксису специфікації. Часто вони були

обмежувальними в нотаціях, які вони підтримували і їх було важко налаштувати, і це було не зручно через постійне позначенням тимчасових невідповідностей. Таким чином, незважаючи на те, що інструменти CASE досягли великих успіхів у підтримці розробки програмного забезпечення, вони все ще залишають бажати кращого, що стосується їх зручності використання, гнучкості та охоплення діяльності розробки [22].

Ще одним додатковим питанням, яке стає все більш критичним для успішного впровадження технології інструментів CASE при розробці програмного забезпечення є інтеграція інструментів. Ідея мати інтегроване середовище розробки, заповнене інструментами CASE або кількома автономними інструментами CASE, що працюють разом, розглядається як фундаментальна і бажана мета. Це пояснюється тим, що розробники програмного забезпечення повинні мати в своєму розпорядженні різноманітні інструменти і мати можливість вибирати конфігурацію інструментів, яка відповідає їхнім вимогам, а також мати можливість використовувати їх окремо або разом в залежності від вимог. Багато складнощів при розробці великих систем виникають через нездатність інструментів або інформаційних систем взаємодіяти одна з одною, наприклад, обмінюватися або ділитися даними.

Інтегровані середовища підтримки проектів (IPSE). Інтегроване середовище підтримки проекту надає набір засобів аналізу, програмування та керування, які охоплюють увесь життєвий цикл розробки програмного забезпечення. Ці інструменти підтримують ще один метод розробки програмного забезпечення, об’єднаний IPSE. Мірою інтеграції інструментів є «ступінь узгодженості інструментів» [23] і це властивість зв’язків, визначених між інструментами. IPSE забезпечує структуру та механізми для визначення та реалізації цих інтеграційних відносин, однак інтеграція інструментів є необхідною, але недостатньою компонентною IPSE. IPSE — це інфраструктура для надання загальних послуг, таких як загальний інтерфейс користувача, загальна база даних або система керування об’єктами,

а також загальний набір інструментів для забезпечення спільної функціональності між інтегрованими інструментами. Ці послуги важливі не лише для забезпечення інтеграції інструментів, а й для інтеграції інтерфейсів, процесів, команди та управління [24]. Розрізняють однорідні (закриті) IPSE, в якому багато інструментів CASE інтегруються відносно легко однією організацією, що створює IPSE, і гетерогенні (відкриті) IPSE, в якому інструменти від різних постачальників ускладнюють інтеграцію.

IPSE іноді називають середовищем, заснованим на методах, на відміну від середовища підтримки програмування (сфера останнього є більш обмеженою з точки зору охоплення життєвого циклу). Це пояснюється тим, що IPSE може підтримувати один або більше методів розробки програмного забезпечення, а не лише програмування. У цьому контексті IPSE базується на конкретній моделі всього або частини життєвого циклу розробки програмного забезпечення. Прикладами IPSE є SIGMA, PACT, ECLIPSE, ARCADIA, тощо.

Середовище UNIX є одним із найвідоміших середовищ підтримки відкритого програмування, хоча вона виникла як операційна система, але розширилася і містить багато механізмів інтеграції та інструментів, які виправдовують її класифікацію як IPSE. UNIX надає примітивний текстовий рядковий інтерфейс, але підтримує низку інструментів розробки програм, включаючи різноманітні редактори, інструменти управління конфігурацією та контролю версій та багато іншого.

Існує ряд різних типів середовищ розробки програмного забезпечення, які забезпечують піднабір функціональних можливостей або охоплення життєвого циклу повномасштабного IPSE. До них належать мовно-орієнтовані середовища які є середовищами підтримки програмування побудовані на конкретній мові програмування та використовують переваги механізмів структурування цієї мови. Приклади таких середовищ є Smalltalk, Interlisp та ін.

Загальнодоступний інструментальний інтерфейс (PTI) — це набір примітивів інтерфейсу, які можуть використовуватися розробниками інструментів і середовищ, що визначають базові засоби інтеграції, такі як керування об’єктами та процесами та комунікації інструментів [25]. Одним із PTI, на основі якого було побудовано багато IPSE є середовище PCTE [25]. Завдання PCTE полягає в тому, щоб забезпечити стандартний рівень на якому постачальники програмного забезпечення можуть створювати та продавати свої інструменти. PCTE надає послуги майже в тому ж стилі, що й інструментальні послуги UNIX із системою керування інтерфейсом користувача (UIMS) для побудови інтерфейсів користувача та керування ними та системою керування об’єктами (OMS) для керування даними (модель даних якої базується на сутності) - модель відносин.

Клас інструментів, відомих як генератори середовища, також є областю все більшого дослідження. Вони генерують мовні або прикладні середовища, як правило, з якогось формального опису. Приклади таких інструментів в якому середовища програмування генеруються з визначень граматики і структурують трансформаційний підхід до генерації специфічних для програми середовищ, описаний у [26].

Технологія Meta-CASE. Інструменти Meta-CASE — це програмні засоби, які підтримують проектування та генерацію інструментів CASE. Генератори середовища, описані вище, є одним із таких інструментів. Вони приймають формальний опис мови чи програми як вхідні дані та генерують один або кілька інструментів для підтримки цієї мови чи програми. Інструменти Meta-CASE для деяких програм (також звані генераторами програм) є досить поширеними. До них належать компілятори, такі як Yacc, генератори коду (Tags), а також мовні системи такі як COBOL і Oracle [27]. Інструменти Meta-CASE, які генерують CASE-інструменти для підтримки методів розробки програмного забезпечення іноді називаютьться методами мета-CASE-інструментів [28].

На рисунку 1.1 наведено схематичний огляд середовища meta-CASE для загального методу. На рисунку припускається, що «визначення методу» достатньо для реалізації інструменту, що підтримує цей метод. На практиці також необхідна певна інформація про інструмент (наприклад, визначення інтерфейсу користувача), перш ніж генератор мета-CASE інструменту зможе створити інструмент CASE, який підтримує визначений метод.

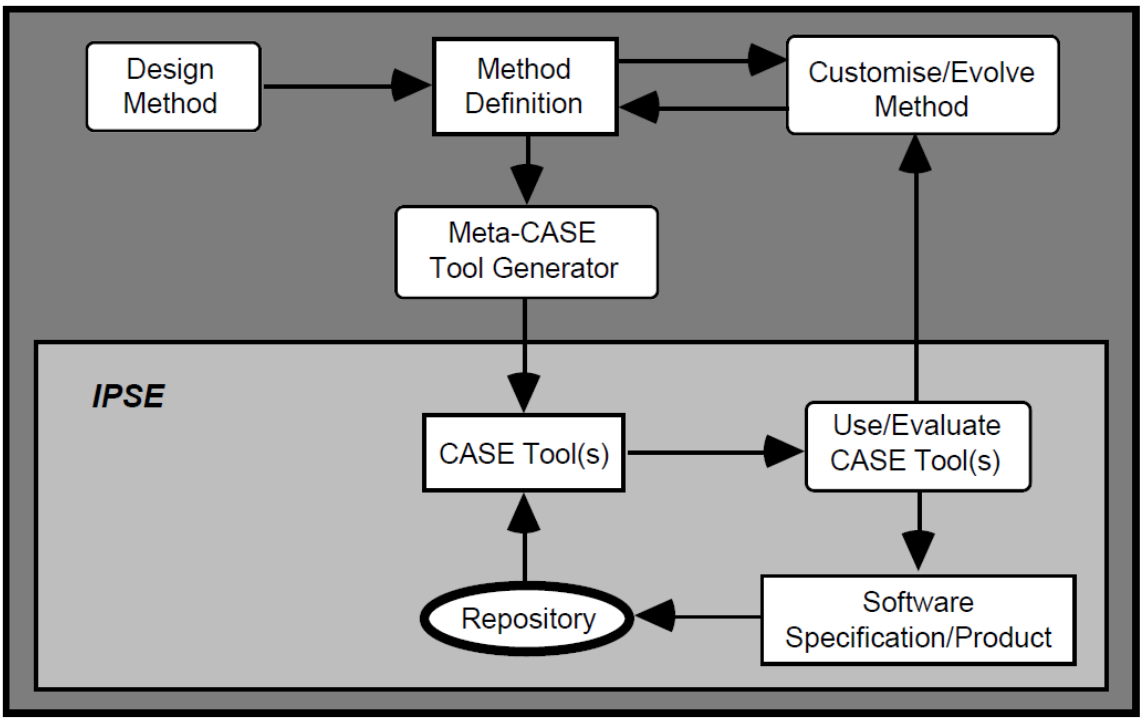


Рисунок 1.1 – Загальне середовище мета-CASE на основі методів

На рисунку 1.1. прямокутники із заокругленими кутами представляють діяльність або процеси, входи та результати яких представлені стрілками. Звичайні прямокутники представляють артефакти, які є виходами, створеними діяльністю чи процесами або входами до них. Овал із товстою рамкою позначає сховище та його систему керування даними або об’єктами.

Мета-CASE-інструменти методу, які ефективно створюють придатні для використання набори інструментів CASE, спираються на ряд ключових факторів і технологій. Метод, для якого необхідна підтримка інструменту, має бути добре зрозумілим і задокументованим, щоб формальний або принаймні

точний опис цього методу міг бути створений і використаний як вхідні дані для мета-CASE інструменту. Хороший meta CASE інструмент також полегшує аналіз, проектування та опис методу, повне та формальне визначення якого може бути не відомими. У цьому налаштуванні інструмент meta-CASE є допоміжним інструментом для розробника методів або інженера і тому його називають інструментом автоматизованого методу розробки (CAME) [29].

Інструменти Meta-CASE також повинні документувати визначення, як правило, графічні нотації та їхні зв’язки. Перевага мета-CASE-інструментів полягає в їх здатності специфікувати та генерувати себе, тобто, як і компілятор, мета-CASE-інструмент також повинен мати можливість самозавантажуватися.

Одним із прикладів комерційного мета-CASE інструменту, який був протестований таким чином є IPSYS Tool Builder's Kit (TBK) [30]. TBK — це інтегрована колекція загальних інструментів і бібліотек функцій, які можна використовувати для визначення методів розробки програмного забезпечення та створення якісних наборів інструментів CASE для їх підтримки. Він дозволяє розробнику інструменту визначати синтаксис і семантику різних нотацій, що застосовуються методом, забезпечує інтерфейс бази даних на основі функціональної моделі даних, яка може бути використана для визначення моделі даних методу (і його продуктів), і включає в себе UIMS на основі OSF/Motif для визначення представлення введених користувачами та виведених із створених інструментів.

Деякі інші мета-CASE-системи також комерційно доступні. До них належать Excelerator, MetaEdit, Paradigm Plus та ObjectMaker [31].

Metaview є багатофункціональним дослідницьким проектом, однією з цілей якого є створення «метасистеми» для опису та автоматичної генерації користувацьких середовищ розробки програмного забезпечення. Архітектура Metaview складається з мета-рівня, рівня середовища та рівня користувача. На мета-рівні визначається мета-модель, яка є достатньо виразною, щоб

підтримувати опис великого класу методів розробки. На рівні середовища визначення середовища, створене на мета-рівні, «обробляється» для створення конкретної конфігурації інструменту. Потім інструменти, створені на рівні середовища, готові до використання розробниками (користувачами) на рівні користувача архітектури.

Підводячи підсумок, мета-CASE-інструменти мають ряд застосувань у контексті розробки методів і розробки CASE-інструментів. Інструмент meta-CASE автоматизує розробку інструментів CASE для розробка програмного забезпечення на основі методів.

Висновки до розділу 1

В даному розділі описується контекст розробки програмного забезпечення роботи. У цьому контексті досліджується роль комп'ютерної підтримки інструментів. Розглянуто та класифіковано методи та інструменти розробки програмних систем.

Інструменти CASE, що були проаналізовані, повністю або частково підтримують метод розробки програмного забезпечення. Часто, намагаючись зробити ці інструменти більш загальними та придатними для повторного використання, вони створюються без точної підтримки пов'язаного з ними методу. IPSE намагаються забезпечити засоби інтеграції інструментів, а деякі IPSE дозволяють розробникам налаштувати процес, за допомогою якого ці інструменти викликаються та використовуються. Проте деталізація таких процесів загалом є надто неточною, щоб забезпечити корисні артефакти щодо методів (це на рівні виклику інструменту, а не на рівні об'єктів, якими маніпулюють ці інструменти).

Технологія інструментів Meta-CASE вписується між інструментами CASE та IPSE, але інструменти meta-CASE часто є автономними. Вони вимагають точних визначень методів як вихідних даних, що робить їх корисними інструментами для визначення процесів розробки та

стандартизації використання нотацій в організаціях. І навпаки, їх також можна використовувати для налаштування процесів і нотацій відповідно до конкретних потреб, а тому налаштування методу проявляється в інструментах, які генерують ці мета-інструменти CASE.

РОЗДІЛ 2. КЛАСИФІКАЦІЯ ТА СТРУКТУРИЗАЦІЯ ПРОГРАМНИХ МЕТОДІВ В КОНТЕКСТІ ЗАДАЧ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Дослідження концепцій інтеграції та агрегації в процесах розробки програмного забезпечення

Складна система визначається як така, складається з великої кількості частин, які взаємодіють у певний спосіб. У таких системах ціле більше, ніж сума частин. Припускається, що складні системи мають ієрархічну структуру, і тому їх краще зрозуміти, якщо розкласти на складові частини. Така декомпозиція є основою для принципу поділу розробки.

Розділення під час проектування - це діяльність, за допомогою якої складна система розкладається за різними критеріями на простіші одиниці. Мета такої декомпозиції полягає в тому, щоб мати можливість розглянути лише ті питання чи критерії, які представляють інтерес, ігноруючи інші, які не мають відношення. Наприклад, можна відокремити розробку функцій інтерфейсу користувача пакета програмного забезпечення від розробки його обчислювальних функцій.

Існує багато можливих критеріїв для поділу проблем під час розробки програмних систем. Одним із поширених критеріїв є час.

Часовий поділ завдань розкладає процес розробки на часові інтервали, протягом яких виконуються різні дії. Тимчасове розділення також може бути корисним для впорядкування діяльності. Наприклад, аналіз вимог зазвичай відокремлюється від впровадження системи та виконується до нього. Модель життєвого циклу водоспаду базується на такому часовому розділенні.

Ще один критерій поділу базується на просторових міркуваннях. Наприклад, розробку програмного забезпечення можна розкласти на дії, що виконуються у фізично різних місцях.

Розділення за якостями також поширене. Наприклад, можна окремо розглянути ефективність і правильність певної програми.

Складність проекту розробки програмного забезпечення також може бути зменшена шляхом розділення різних видів програмної інженерії, які задіяні. Наприклад, розробники програмного забезпечення часто створюють моделі даних і процесів складної системи та розглядають їх окремо. Такий вид поділу перегляду часто використовується, коли різні частини або аспекти складної системи потрібно розглядати окремо. П -парадигма застосовує такий поділ проблем, щоб розділити складний опис програмного забезпечення на менші (більш зрозумілі) часткові описи (так звані перегляди). Потім вони накладаються, щоб сформувати повний опис.

Для парадигми розробки програмного забезпечення стає все більш можливим критерії для поділу. Це особливо актуально, коли багатопарадигмальні системи розробки програмного забезпечення стають все більш поширеними. Парадигма відноситься до підходу, який характеризується поглядом і представленням оточення в якому ця система працює. Приклади специфікації програмного забезпечення та парадигм програмування включають функціональну, логічну, об’єктно-орієнтовану та імперативну парадигми. Все більш складні програмні системи можуть вимагати розробки їх різних аспектів або компонентів з використанням різних парадигм. У таких випадках поняття парадигми є корисним грубим критерієм для розділення проблем і декомпозиції цих систем і процесів їх розвитку.

Підсумовуючи, поділ проблем стає все більш важливим інструментом для вирішення складності багатьох сучасних програмних систем. Він ґрунтується на розкладанні проблеми або системи на її складові компоненти, які, наскільки це можливо, не пов’язані між собою або принаймні слабко пов’язані. Ці простіші компоненти потім можуть бути розглянуті (розроблені, проаналізовані, декомпозиційні тощо...) незалежно, різними учасниками розробки, у різний час і у різних місцях тощо.

Інтеграція — це процес створення чогось цілого. У розробці програмного забезпечення можливості для інтеграції є досить великими як і розділення завдань, воно може ґрунтуватися на низці різних критеріїв. Наприклад, можна інтегрувати або поєднати діяльність різних учасників розробки щоб вони були синхронізовані та скоординовані. Як альтернатива, можна об’єднати окремі підпродукти різних видів діяльності з розробки, щоб отримати єдиний результат (наприклад, програмну систему).

До видів інтеграції відносяться метод, інструмент, вид і системна інтеграція. Усі вони підпадають під загальну класифікацію інтеграції знань інженерії програмного забезпечення та слабо пов’язані між собою, що схематично показано на рисунку 2.1.

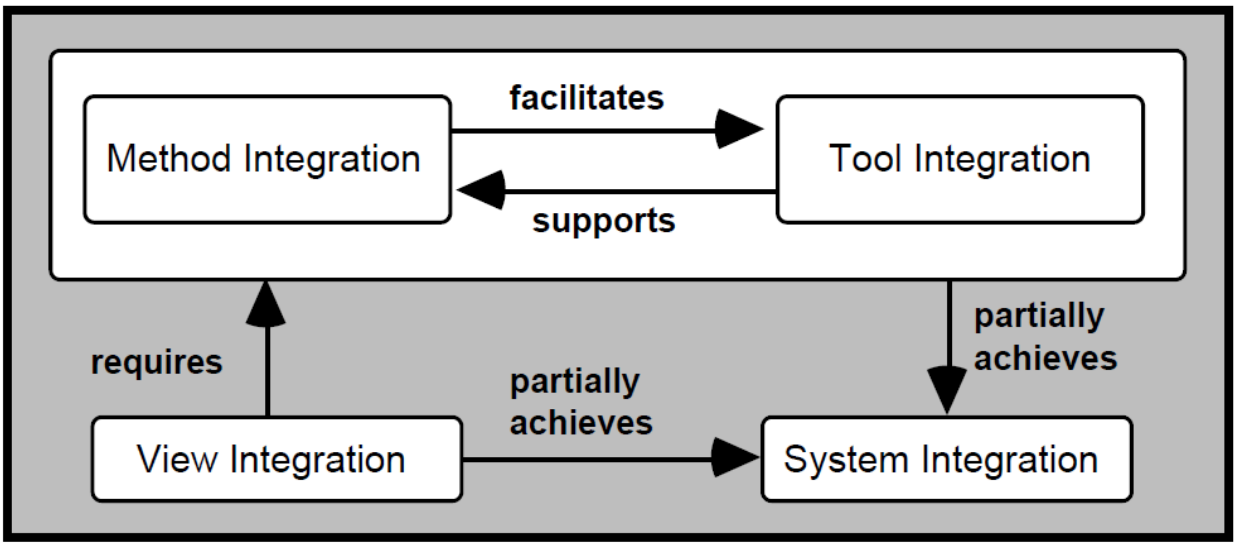


Рисунок 2.1 – Методика інтеграції знань програмної інженерії

Інтеграція інструментів у контексті автоматизованої розробки програмного забезпечення відноситься до комбінації двох або більше (інструментів розробки програмного забезпечення таким чином, щоб ці інструменти потім могли обмінюватися інформацією для досягнення певної мети розробки. Структура інтеграції інструментів — це інфраструктура, що містить механізми для досягнення інтеграції [32]. Структури інтеграції інструментів забезпечують високорівневі архітектури, які позиціонують і

пов’язують різні компоненти інтеграції інструментів, які зазвичай включають принаймні систему керування об’єктами, агента інтеграції (наприклад, протокол зв’язку інструментів) і самі інструменти.

Інтеграція інструментів зазвичай є основною метою інтегрованого середовища підтримки проекту (IPSE) [33]. На жаль, це дещо змінило більш фундаментальну мету IPSE – забезпечення інтегрованої інструментальної підтримки для розробки програмного забезпечення на основі методів. Таким чином, увага часто була зосереджена на структурах, інфраструктурах, протоколах, механізмах і стандартах для інтеграції інструментів, лише з другорядною вимогою щодо підтримки методів розробки програмного забезпечення. Це контрастує з більш бажаним підходом розробки структур і методів для інтеграції методів, які потім можуть бути використані як основа для підтримки інструментів та інтеграції [34].

Інтеграція методів [35] фокусується на вивченні та синтезі багатьох методів, які мають індивідуальні та колективні особливості і роблять їх привабливою комбінацією для розробки конкретних (програмних) систем. Інтеграція інструментів у цьому контексті виникає лише після запитань импу «які методи слід інтегрувати?» і «як ці методи використовуватимуться разом?». Метою інтеграції методу та інструменту є створення інтегрованої системи програмного забезпечення. Тому системна інтеграція в цьому контексті стосується продукту розробки, а не самого процесу. Оскільки система складається з кількох компонентів, з’єднаних певним чином, системна інтеграція – це процес встановлення цих зв’язків. Системна інтеграція є бажаною при розробці програмного забезпечення з багатьох причин. До них належать вимоги замовника щодо єдиного інтегрованого результату, а також тому, що інтегровані системи мають «вихідні властивості», якими не обов’язково володіють їхні складові частини.

Вимоги великих, складних систем часто суперечливі і їх майже завжди важко зрозуміти, виявити та визначити точно та повністю. Клієнти можуть бути не впевнені щодо своїх системних вимог, а аналітики можуть по-різному

розуміти або інтерпретувати ці вимоги та проблемну область, у якій буде працювати потрібна система. Тому підходи, які підтримують численні точки зору в інженерії вимог, є особливо корисними, навіть необхідними. Репрезентативна вибірка таких підходів наведена в таблиці 2.1.

Таблиця 2.1

Підходи в розробці вимог

Approach	Definition of viewpoint
SA	An expression of interest
CORE	An information processing entity
Viewpoint resolution	A standing or mental position held by an individual examining or observing a universe of discourse
VOA	A service recipient
Goal-directed approaches	A domain model with associated objectives (“a belief”)

Одне з найперших застосувань точок зору в розробці вимог було в структурованому аналізі (SA). В SA точка зору виражає інтерес до певного аспекту системи і пов’язана з контекстом та метою, які разом описують орієнтацію моделі SA. Метод SADT заснований на SA, розглядає точку зору як концептуальне поняття, а не як формальний засіб для фіксації та вираження системних вимог. Однак метод SADT має хорошу нотацію та багато евристик для побудови кількох моделей системи, які, у свою чергу, підтримують кілька точок зору в процесі специфікації вимог.

Метод контрольованих вимог (CORE) чітко визначає точки зору на ранніх стадіях процесу виявлення та специфікації вимог, ґрунтуючись на систематичному визначенні проблеми, у якому також визначаються повноваження клієнта. Точки зору в CORE представляють об’єкти обробки інформації, які систематично аналізуються в міру розгортання методу. Однак точки зору в CORE є ортогональними, тобто не перекриваються. Це

нереалістично і незадовільно, якщо потрібно підтримувати кілька точок зору або поглядів на реальну систему.

Існує підхід до визначення, аналізу та визначення вимог, орієнтованого на VOA. Цей підхід використовує поняття точок зору для представлення сутностей, зовнішніх щодо системи, що аналізується, які можуть існувати без присутності системи. Точками зору в підході VOA є одержувачі послуг, які можуть надавати системі дані або керуючу інформацію, необхідну для надання послуг.

Хоча послуги, отримані точками огляду, представляють функціональні вимоги, ці послуги також можуть бути пов’язані з обмеженнями, які представляють нефункціональні вимоги. Таким чином, VOA забезпечує структуру для інтеграції функціональних і нефункціональних вимог, полегшуючи раннє виявлення та вирішення невідповідностей, коли вони виникають.

Інші підходи до виявлення вимог менш чітко використовують термін точки зору, але, тим не менш, посилаються на нього. Наприклад, у [36] пропонується цілеспрямований підхід до виявлення вимог, у якому системні вимоги будуються шляхом визначення цілей, які потім можуть бути пов’язані з агентами, їхніми діями, ролями, відповідальністю тощо. Цілі, визначені таким чином, можуть бути використані для забезпечення основи, на якій можуть бути вирішені конфлікти та узгоджені численні точки зору.

Підходи до інтеграції методів включають:

1) інтеграцію за загальними ознаками, такими як загальна модель, що лежить в її основі;

2) обмеження дослідження сумісними підмножинами методів, такими як мовні підмножини;

3) розширення одного «основного методу» за допомогою ідей або взятих функцій від інших «додаткових методів»;

4) посилення основного методу шляхом заміни або перевизначення деяких його ідей іншими, взятими з додаткових методів.

Однак на практиці інтеграція методів є випадковою і є побічним ефектом інтеграції інструментів, оскільки інтеграція інструментів є реалізацією інтеграції методів. Як наслідок, значна частина дослідницької роботи з інтеграції стосується структур для досягнення інструментальної інтеграції та забезпечення механізмів взаємодії та обміну інформацією між інструментами. Хоча такі механізми актуальні в контексті інтеграції методів, вони фундаментально орієнтовані на реалізацію та часто не вирішують більш концептуальні та семантичні питання інтеграції методів.

2.2 Фреймворки, архітектура та механізми інтеграції процесів розробки

Структура інтеграції визначає налаштування або контекст, у якому інструменти, методи та програмні продукти пов’язані та можуть бути досліджені. Часто також розглядається методика застосування інтеграційного підходу. З іншого боку, архітектура інтеграції описує більш детально частини фреймворку з деякими вказівками на те, як вони взаємодіють між собою. У цьому контексті терміни «фреймворк» і «архітектура» часто використовуються як синоніми.

Однак термін «архітектура програмного забезпечення» також використовується для позначення високорівневого опису дизайну програмної системи, вираженого у відповідній нотації

В архітектурі Eureka Software Factory (ESF) [36] багаторівневе уявлення про комунікаційні потоки розглядається як частина концептуальної еталонної моделі CoRe. Комунікація в ESF здійснюється через одну «програмну шину», хоча частини архітектури (розробники, інструменти та платформи) отримують пряме спілкування «один-на-один» (рис. 2.2).

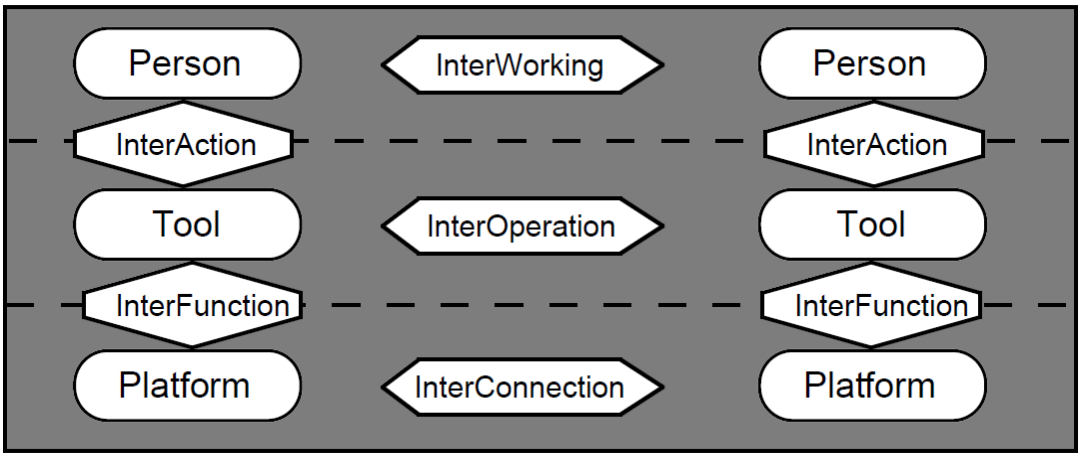


Рисунок 2.2 – Зв’язок через «програмну шину» ESF

Хоча архітектура ESF є кроком до децентралізації, її програмна шина є вузьким місцем зв’язку, яке зазвичай пов’язане з інтеграційними архітектурами, які мають суворо централізований контроль. Крім того, хоча архітектура ESF описує взаємодію між людьми, платформами та інструментами, вона конкретно не стосується взаємодії методів.

Загалом, архітектури централізованої інтеграції (рисунок 2.3a) покладаються на загальну («централізовану») сутність (наприклад, репозиторій або сервер) для полегшення інтеграції. Архітектури децентралізованої інтеграції (рисунок 2.3b) покладаються на спеціальну інтеграцію один до одного складових компонентів.

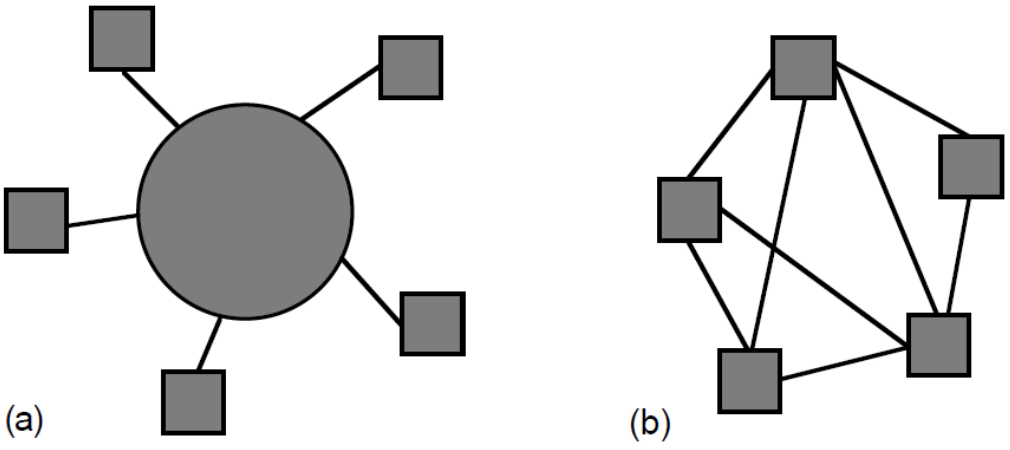


Рисунок 2.3 – Централізована та децентралізована інтеграційні архітектури

Між цими двома сутностями існує цілий ряд архітектур. Наприклад, деякі фізично централізовані бази даних логічно розподілені (тобто вони можуть «виглядати» різними частинами організації як різні бази даних), а деякі розподілені системи мають централізоване керування (наприклад, у системі клієнт-сервер розподілені клієнти працюють на централізованому сервері для контролю).

Проблемою в інтеграції інструментів є загальна відсутність згоди розробників інструментів щодо відповідних механізмів інтеграції, рівнів інтеграції та стандартів для представлення, даних і контролю інтеграції. Інтеграція вирішує потребу мати узгоджене та послідовне візуальне уявлення про інформацію, створену різними інструментами. Зазвичай це передбачає певну стандартизацію користувацьких інтерфейсів інструментів, наприклад, шляхом запуску цих інструментів в одній системі.

Інтеграція даних зазвичай відноситься до процесу створення узгодженого масиву інформації, створеного різними інструментами. Це дозволяє синтаксис і семантику даних, створених одним інструментом, розуміти та використовувати іншим. Інтеграція керування, з іншого боку, — це процес координації та синхронізації операцій і функцій різних інструментів для їхньої спільної роботи. Інтеграцію керування іноді також називають інтеграцією процесу, але, це лише її підмножина. Інтеграція процесів, як правило, охоплює процеси всієї організації, які не обов’язково орієнтовані на інструменти.

Незважаючи на те, що інтеграція даних, керування та представлення є різними аспектами інтеграції (інструментів), їх потрібно розглядати спільно, щоб досягти кінцевої спільної мети побудови програмної системи. Цій меті служить інтеграційна структура або архітектура. Підтримка децентралізації в інтеграційних архітектурах є важливою через все більш розподілений характер розробки програмного забезпечення. На жаль, сховища, які зазвичай використовуються, не підходять для підтримки розподіленої розробки. Спільні репозиторії базуються на єдиних (монолітних) метамоделях або

схемах, які важко поширювати, розуміти, підтримувати та розширювати. Тому вони можуть перетворитися на вузькі місця, а не на інтеграційні агенти середовищ розробки програмного забезпечення.

Проект ESPRIT [38] розглядає інтеграцію методів досить детально. Приймаючи різні аспекти інтеграції інструментів, визначені вище, також визначено два аспекти інтеграції методів. Це внутрішньопроцесна інтеграція, яка стосується поєднання різних методів і міжпроцесна інтеграція, яка стосується поєднання керування конфігурацією з системною інтеграцією.

Таким чином, відбувається зосередження на композиції методів у різних процесах розробки та між ними. Це досліджується за допомогою ряду різних прикладів, які намагаються виконати попарну інтеграцію різних методів. Результатом такої інтеграції методів є висвітлення низки проблемних питань, корисних вказівок. Вони включають потребу у вирішенні термінологічних суперечностей між методами, ідентифікації різних ієрархій, які розгортають різні методи, виявленні та вирішенні невідповідностей у продуктах методів та інструментів, а також надання або використання інструментів перетворення для відстеження та перевірки узгодженості.

Пропонується системний підхід до інтеграції методу, який включає наступні кроки, які необхідно виконати для створення «нового» інтегрованого методу:

1. Визначити цільовий процес інтегрованого методу, включаючи визначення кроків розробки (процедур розробки) і порядок цих кроків.

2. Вибрати методи, які потрібно інтегрувати, в ідеалі, тому що вони підтримують цільовий процес, але часто з прагматичних причин (таких як наявність підтримки інструментів або досвіду).

3. Визначити модель, що лежить в основі інтегрованого методу, щоб ідентифікувати класи об’єктів, які представляють, маніпулюють і аналізують цей метод. Базова модель методу забезпечує представлення продукту(ів), створених цим методом.

4. Визначити мовні відображення між (підмножинами) мов «оригінальних» методів. Оскільки базову модель інтегрованого методу вже визначено (на кроці 1), потенційно велика кількість попарних відображень ((n 2 + n)/2 для n мов) зводиться до n відображень (тобто одне відображення з кожну мову в базову модель). Звичайно, це припускає, що потрібні попарні відображення між усіма вихідними мовами, що не завжди так.

5. Надати вказівки для користувача щодо використання нового інтегрованого методу. В ідеалі більша частина настановної інформації повинна бути отримана з визначення процесу розробки (тобто з моделі процесу цього процесу), але тематичні дослідження, приклади та евристики також є важливими джерелами такого методичного керівництва. Однак інтеграція керівництва, яке забезпечується багатьма методами, залишається проблематичною.

Хоча наведені вище кроки описують підхід до інтеграції методів, також необхідна структура інтеграції методів, щоб полегшити поєднання будь-яких двох або більше методів таким чином, щоб вищезазначені проблеми (плюс обробка неузгодженості та підтримка інструментів) могли бути розглянуті в систематичний спосіб. Тому пропонується використовувати «мета-метод» для інтеграції методу, хоча побудова такого методу може бути дуже складною (і майже рівнозначна пошукам «універсального» методу). Потреба в такій організаційній структурі також підтверджена і пропонує структуру для інтегрованого CASE. Однак структура, як і багато інших, насамперед стосується інтеграції інструментів, а не інтеграції методів.

У той час як фреймворки та архітектури забезпечують контекст, у якому можна досягти інтеграції, протоколи та механізми є засобами для досягнення такої інтеграції. Протокол визначає правила та дії, за допомогою яких об’єкти взаємодіють, тоді як механізм реалізує такий протокол. Оскільки інтеграція — це процес об’єднання двох або більше об’єктів (наприклад, інструментів, методів, систем тощо), для виконання цього поєднання необхідні комунікаційні протоколи та механізми.

Протоколи зв'язку між взаємодіючими об'єктами відрізняються за своїми цілями. Це накладає різні вимоги до деталізації та синхронізації інформації, якою обмінюється.

Добре відомий протокол, розгорнутий у багатьох мультиагентних системах, — це узгодження з використанням контрактних мереж. У цьому протоколі встановлюється явна угода (контракт) - після процесу торгів - між вузлом менеджера, який генерує завдання та іншим вузлом підрядника, який бажає виконати це завдання. Потім цей контракт використовується як основа для приватного двостороннього зв’язку (передачі інформації), доки завдання не буде виконано до кінця підрядником.

В [39] визначають протокол взаємодії в контексті розподілених систем як «набір правил і угод, яких модуль повинен дотримуватися, щоб спілкуватися або синхронізуватися з іншим модулем». Виклик віддаленої процедури (RPC) є одним із таких протоколів, який використовується в багатьох розподілених системах, що розгортають архітектуру клієнт-сервер. Цей протокол ґрунтується на передачі повідомлень між вузлами, коли клієнт надсилає назву процедури та параметри процедури на сервер, а потім чекає на відповідь.

Сервер з іншого боку чекає на отримання запиту, обробляє його локально, коли він надходить, а потім повертає результат клієнту. Клієнт у цьому протоколі знає про сервер і делегує цьому серверу виконання певних завдань. Протокол RPC можна використовувати в контексті інших спільних протоколів, які мають більш загальні спільні цілі.

Загалом, клієнт-серверна архітектура класифікує протоколи зв’язку за двома широкими заголовками: пересилання даних (наприклад, кешування) та пересилання функцій (наприклад, віддалена служба). У протоколі доставки функцій клієнт надсилає дані та код операції на сервер, а потім, можливо, чекає на результат. Реалізації цього типу протоколу широко поширені, але можуть страждати від накладних витрат зв’язку та потенційно високого мережевого трафіку. З іншого боку, у протоколі доставки даних клієнт

запитує/отримує інформацію з сервера та виконує операції з цією інформацією локально. Реалізації цього протоколу можуть бути неефективними, якщо є висока частота викликів доступу/операцій, що може знадобитися, якщо існує висока швидкість зміни отриманої інформації сервера.

Архітектури спільної роботи також розгортають різноманітні протоколи зв’язку. Наприклад, архітектура PACT пропонує інфраструктуру для кластерів агентів (програм, які інкапсулюють інженерні інструменти), щоб взаємодіяти через фасилітаторів, які «перекладають знання про інструменти на стандартну мову обміну знаннями та з неї». Це так звана «об'єднана архітектура». Альтернатива цьому підходу продемонстрована в архітектурі MIT Dice, в якій мережа агентів спілкується через спільний робочий простір, який називається «чорна дошка», агент у Dice — це комбінація користувача та комп’ютера. Відповідно багатопарадигмальне програмування також може потребувати комунікаційних протоколів для інтеграції.

Мультипарадигмальні системи вимагають координації, взаємодії та синхронізації ряду програмних фрагментів, які разом забезпечують загальну бажану функціональність системи. На відміну від багатоагентних систем, у яких конкретною метою є цілеспрямоване кооперативне вирішення проблем, багатопарадигмальні системи програмування мають більшу схожість із системами з кількома представленнями, у яких численні фрагменти програми розглядаються як погляди на домен проблеми або рішення. У цьому випадку протоколи зв’язку підтримують такі види діяльності, як сумісність, трансформація, злиття і композиція поглядів.

Таблиця 2.3 перераховує різні методи інтеграції кількох часткових специфікацій. Ці методи можуть використовуватися як окремі складові інтеграції методів і не обов’язково є взаємовиключними.

Таблиця 2.3

Представлення методів інтеграції

Technique	Example mechanisms deployed
Composition	1. Conjunction 2. Amalgamation
Interoperability	1. Specification level 2. Programming level 3. User level (negotiation); e.g.: • contract nets • using domain goals
Merging	1. Operation based 2. Evolutionary transformations
Transformation	1. Syntax integration 2. Meta-system approach

Отже, формальна композиція – це процес об’єднання формальних часткових специфікацій, які описують різні або схожі погляди на систему.

У контексті багатонапрямленої розробки програмного забезпечення, процес агрегації – це процес об’єднання результатів діяльності паралельної розробки. Цього особливо важко досягти, якщо дії заважають (наприклад, під час розподіленої, кооперативної розробки) або результати цих дій збігаються (наприклад, контроль версій). Багато методів агрегації також використовують різні види перетворень.

2.3 Дослідження концепції побудови ViewPoints Framework

У розробці більшості великих і складних систем бере участь багато людей, кожен з яких має власну точку зору на систему, що визначається їхніми навичками, відповідальністю, знаннями та досвідом. Це особливо вірно для композитних систем, які розгортають ряд різних технологій.

Різні точки зору тих, хто бере участь у процесі розробки таких систем, перетинаються та накладаються, породжуючи вимогу перевірки узгодженості та координації. Але перетин, зовсім не очевидний, оскільки знання в межах кожної точки зору представлені по-різному. Крім того, оскільки розробка може здійснюватися одночасно тими, хто бере участь, різні точки зору можуть перебувати на різних стадіях розробки та можуть підлягати різним стратегіям розвитку.

Проблема того, як організувати та спрямовувати розвиток у цьому середовищі – багато учасників, різноманітні схеми репрезентації, різноманітні знання предметної області, різні стратегії розвитку – все це називається «проблемою кількох точок зору». Хоча це загальна проблема розробки великих систем, ми зосередимося лише на багатонапрямленій розробці програмного забезпечення - підмножині розробки систем.

Для розгляду проблеми кількох точок зору необхідно виконати наступні вимоги:

• структурувати, організувати різні процеси під час розробки програмного забезпечення;

• перевірити узгодженість і невідповідності між різними точками зору, передаючи та перетворюючи інформацію між ними;

• підтримка одночасної спільної розробки в розподіленому середовищі.

Задоволення кожної з цих вимог потребує створення організаційної структури ViewPoints Framework, а саму багатонапрямлену розробку програмного забезпечення в рамках ViewPoints будемо називати ViewPoint-Oriented Software Engineering (VOSE).

Термін «перегляд» (View) і «точка зору» (ViewPoint) використовується по-різному в інженерії програмного забезпечення.

У світі баз даних представлення часто розглядається як проекція або часткове представлення глобального джерела знань. Збіги (які потенційно призводять до невідповідностей) підтримуються централізовано глобальним джерелом знань. Ми використовуємо термін «Точка огляду» з великими

літерами «V» і «P» у верхньому регістрі. Це не слід плутати з системою ViewPoints, описаною в [41], яка є інструментом обґрунтування дизайну на основі гіпертексту.

Однак інженери програмного забезпечення все частіше розглядають представлення як слабко пов’язані джерела знань, які можна модифікувати незалежно.

Загальна створена база даних або представлення відповідає за виявлення збігів і вирішення невідповідностей між цими представленнями. Перегляди в цьому параметрі більше схожі на інструменти, які мають незалежні ідентифікатори, поведінку та стан.

Щоб трактувати представлення як активні сутності (тобто як інструменти з певною функціональністю) потрібно, щоб ці представлення взаємодіяли та обмінювалися інформацією. Однак таке функціонування джерела знань для сприяння цій взаємодії та обміну поступово стає вузьким місцем.

Визначення 1. ViewPoints визначаються як слабозв’язані локально керовані розповсюджувані об’єкти.

Кожна точка огляду (ViewPoint) інкапсулює часткові знання про систему та її область, виражені у відповідній схемі представлення, разом із частковими знаннями про загальний процес розробки. Таким чином, ViewPoint містить три типи знань програмної інженерії:

- знання представлення про нотацію, мову або схему представлення, що використовуються цією точкою огляду;
- знання специфікації щодо конкретного аспекту системи або проблеми, з якою стурбована точка огляду;
- знання процесу розробки програмного забезпечення про те, як розгорнути певне представлення для створення специфікації.

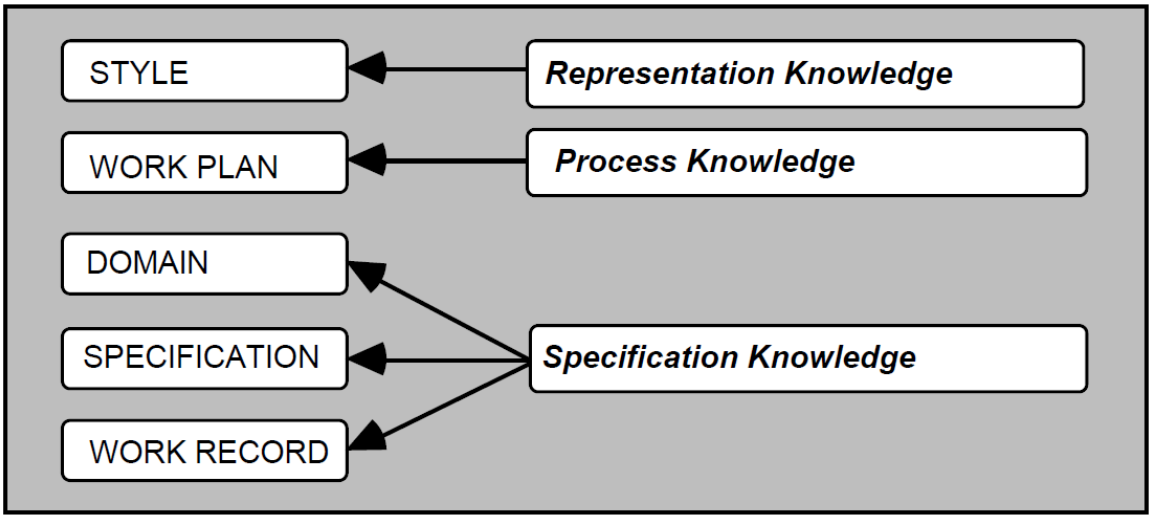


Рисунок 2.4 – Шари / слоти ViewPoint

Різні види знань, які інкапсулює ViewPoint, структуровані та розділені на п’ять шарів (слотів) (рис. 2.4):

1) стиль представлення;

2) план роботи;

3) предметна область;

4) специфікація;

5) інформація про роботу.

Ці шари використовуються для розділення проблем у межах різних видів знань, які вони містять і надають точкам огляду єдину структуру та служать для ідентифікації та розділення різних точок огляду.

Шар “Representation Style” (надалі просто шар стилю) містить визначення схеми представлення, розгорнутої точкою огляду. Таким чином, він містить мета-рівневий опис нотації або мови, що використовується точкою огляду для опису проблемної області цієї точки огляду.

Текстові мови часто визначаються описом граматики BNF, тоді як для графічних нотацій можуть бути більш придатними зв’язки сутностей. Ми використовуємо нотацію на основі об’єктів і зв’язків, які мають типізовані атрибути та значення, щоб описати стиль представлення ViewPoint.

Прикладом об’єкта є «Процес» на діаграмі потоку даних, який у нашому представленні показано на рис. 2.5.

Термін «слот» запозичено з фреймів у штучному інтелекті, де він використовується як контейнер знань або даних і ще називається «змінною фрейму».

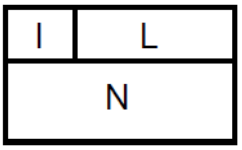
OBJECT: Process		
ATTRIBUTES	**TYPES**	**VALUES**
Name (N)	String	
Identifier (I)	Integer	
Location (L)	String	
Icon	Bitmap	

Рисунок 2.5 – «Процес» на діаграмі потоку даних

З іншого боку, відношення пов’язує два або більше об’єктів (які можуть бути виражені як параметри цього відношення). Наприклад, «Перехід» на діаграмі переходу станів є зв’язком між двома об’єктами «Стан» і може бути визначений таким чином як показано на рис. 2.6.

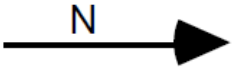
RELATION: Transition (State, State)		
ATTRIBUTES	**TYPES**	**VALUES**
Name (N)	String	
Icon	Bitmap	

Рисунок 2.6 – Приклад об’єднання двох відношень на діаграмі станів

Хоча таке представлення на мета-рівні більше підходить для графічних мов, в принципі його також можна використовувати для текстових мов. Одним із підходів було б ідентифікувати термінали у визначенні BNF і використовувати їх як об’єкти в нашому представленні.

Тоді слот стилю надає базове визначення синтаксису нотації, що використовується ViewPoint. Як розгортається ця нотація, описано в слоті робочого плану.

Слот робочого плану визначає дії та рекомендований порядок цих дій, необхідних для розгортання схеми представлення, визначеної в слоті стилю. Таким чином, робочий план містить низку дій і модель процесу, яка описує обставини, за яких будь-яка з дій може бути виконана. Ми виділяємо п’ять різних видів дій:

• Агрегаційні дії. Це основні дії, необхідні для складання (конструювання) специфікації ViewPoint у визначеній схемі представлення. Їх можна розглядати як список базових дій «редагування» засобами CASE, що підтримує таку ViewPoint. Ці дії включають додавання та видалення об’єктів, а також зв’язування та роз’єднання об’єктів – шляхом додавання та видалення будь-яких зв’язків між цими об’єктами.

• Дії перевірки ViewPoint. Це події, які перевіряють локальну узгодженість специфікацій ViewPoint відповідно до деяких правил. Правила In-ViewPoint по суті визначають семантику стилю представлення, визначаючи обмеження, які повинні дотримуватися, щоб специфікація ViewPoint була «правильно сформованою». Розробники різних методів можуть застосувати різні правила ViewPoint до специфікацій. У функціональній ієрархії, наприклад, правило in-ViewPoint може передбачати, що для будь-якої декомпозованої батьківської функції має бути більше одного дочірнього елемента, тоді як інші розробники методів можуть дозволити декомпозицію батьківської функції на одну дочірню (рис. 2.7). Потім перевірка у ViewPoint визначає чи виконується це правило, тобто перевіряється чи має будь-яка декомпозована функція в ієрархії більше одного дочірнього елемента. Інші дії

перевірки у ViewPoint можуть включати перевірку повторюваних імен функцій, наявність одного кореня для кожної функціональної ієрархії тощо.

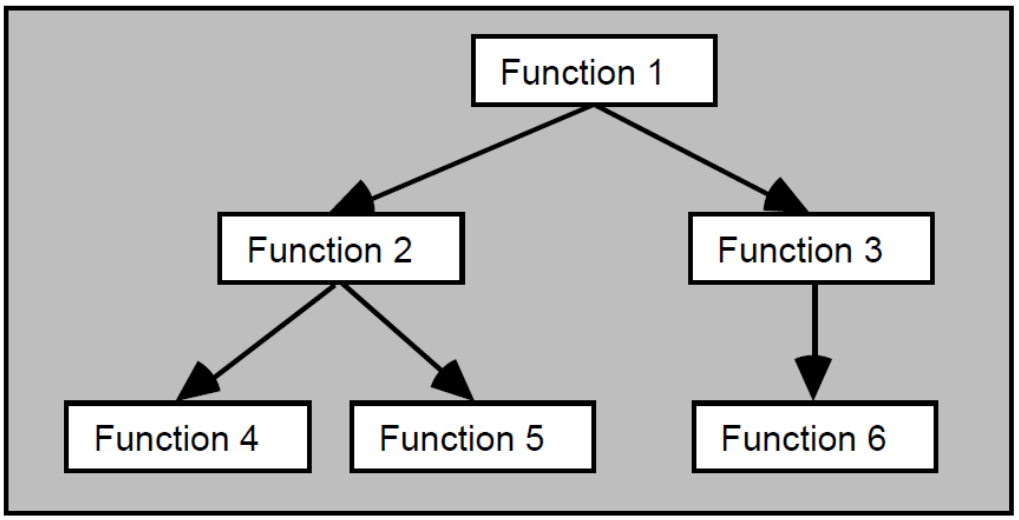


Рисунок 2.7 – Ієрархія функціональної декомпозиції

На рисунку 2.7 дія перевірки у ViewPoint позначатиме невідповідність у цій діаграмі, лише якщо існує правило у ViewPoint, встановлене розробником методу, яке вказує, що одна декомпозиція функції є не вірною. Таке правило виявить, що «Функція 3» розкладається лише на «Функцію 6».

• Дії перевірки Inter-View Point. Це дії, які перевіряють узгодженість між точками перегляду відповідно до деяких правил між точками перегляду. Правила Inter-ViewPoint визначають зв’язки між стилями представлення різних точок огляду і тому описують області перекриття між точками огляду. Перевіряючи правила між точками огляду в чітко визначеному порядку їх також можна використовувати як засіб для синхронізації або координації дій розробки двох різних точок огляду. Цей порядок передбачено локальними моделями процесу ViewPoint. У процесі вираження зв’язків між точками огляду було визначено чотири різні види, перелічені та пояснені на рисунку 2.8.

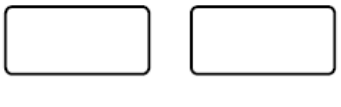
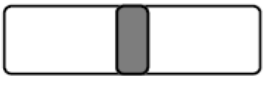
	The two ViewPoints are independent, non-overlapping and unrelated (except in that the method from which they are created requires both ViewPoints to exist - but the ViewPoints themselves, and their contents, are unrelated). For example, a method may require the development of a ViewPoint describing the functional hierarchy of a software system and another ViewPoint documenting the financial resources available to the project.
	The two ViewPoints are non-overlapping, but there is some existential relationship in which the existence of one depends in some way on the existence of the other. For example, the Z method [Spivey 1989] requires that for each Z schema ViewPoint, there is an associated textual description ViewPoint.
	The two ViewPoints are partially overlapping, with a partial specification in one related to a partial specification in the other. This is the most common type of relationship. For example, a source agent in a CORE [11] tabular collection diagram must be a named agent in the agent hierarchy. Relationships between partially overlapping ViewPoints describe syntactic and semantic correspondences between notations and domains of concern respectively.
	The two ViewPoints are totally overlapping - they describe the same domain using the same representation scheme. We may (1) require that any conflicts, discrepancies or inconsistencies be eventually resolved so that the two ViewPoints are made to say the same thing, or (2) accept that the two ViewPoints represent two different “views” of the same domain (e.g., different solutions to the same problem) that require evaluation and a choice to be made between them.

Рисунок 2.8 – Види зв’язків між ViewPoint

• Тригерні дії ViewPoint. Ці дії створюють нові ViewPoint і вони є діями мета-рівня. Щоб зробити структуру ViewPoints повністю розповсюджуваною, ці дії є частиною окремих планів роботи ViewPoint і не обов’язково виконуються централізовано. Таким чином, ViewPoints самі можуть створювати нові ViewPoints «на льоту» зі своїх власних індивідуальних характеристик.

• Дії для подолання неузгодженості. Це дії, які можуть бути виконані за наявності неузгодженості і є підмножиною інших дій робочого плану, перелічених вище. Однак вони можуть включати додаткові дії, які явно не є частиною робочого плану ViewPoint, наприклад дії користувача, необхідні для вирішення конфлікту, консультації експертів або подальшого пошуку фактів і дослідження. Часто це дії, які викликають спеціалізовані інструменти для вирішення конкретних проблем. Дії обробки невідповідностей можна використовувати для обробки невідповідностей як у точках огляду так і між ними, хоча проблеми, пов'язані з обробкою між точками перегляду, набагато складніші. Обробка невідповідності заснована на логіці правила обробки як частина моделі процесу ViewPoint. Ці правила мають загальний вигляд:

INCONSISTENCY implies ACTION

Усі перераховані вище дії робочого плану є діями, які може виконувати розробник ViewPoint. Модель процесу ViewPoint визначає , коли потрібно виконувати будь-яку з цих дій. Іншими словами, модель процесу забезпечує вказівки щодо стратегії розробки специфікації ViewPoint. Демонструючи фреймворк ViewPoints, прийнято наступне позначення для опису локальної стратегії розвитку ViewPoint (рис. 2.9):

{preconditions} ⇒ [agent, action] {postconditions}

Рисунок 2.9 – Позначення для опису локальної стратегії розвитку ViewPoint

І це означає наступне (рис. 2.10):

if the listed *preconditions* hold, then,
if the *agent* performs the *action*, then,
the list of *postconditions* will hold

Рисунок 2.10 – Визначення стратегії розвитку ViewPoint

Крім того, локальне моделювання окремих процесів розробки ViewPoint є кроком до повністю розповсюджуваної структури ViewPoints, у якій також розподіляється «загальний» процес розробки. Фактично, вважається, що поняття загального процесу розробки або моделі процесу не є корисним у цьому контексті. Те, що ми маємо і що нам потрібно підтримувати — це різні погляди на загальний процес. До них належать організаційні, управлінські рішення, розвиток продукту тощо.

2.4 Характеристики, інфраструктура, шаблони та методи побудови ViewPoints Framework

Специфікація системи у структурі ViewPoints — це конфігурація (структурована колекція) ViewPoints (рис. 2.11). Це узгоджується з фактичною практикою специфікації, коли і процес розробки і кінцеві результати є серією артефактів, на відміну від єдиної монолітної специфікації.

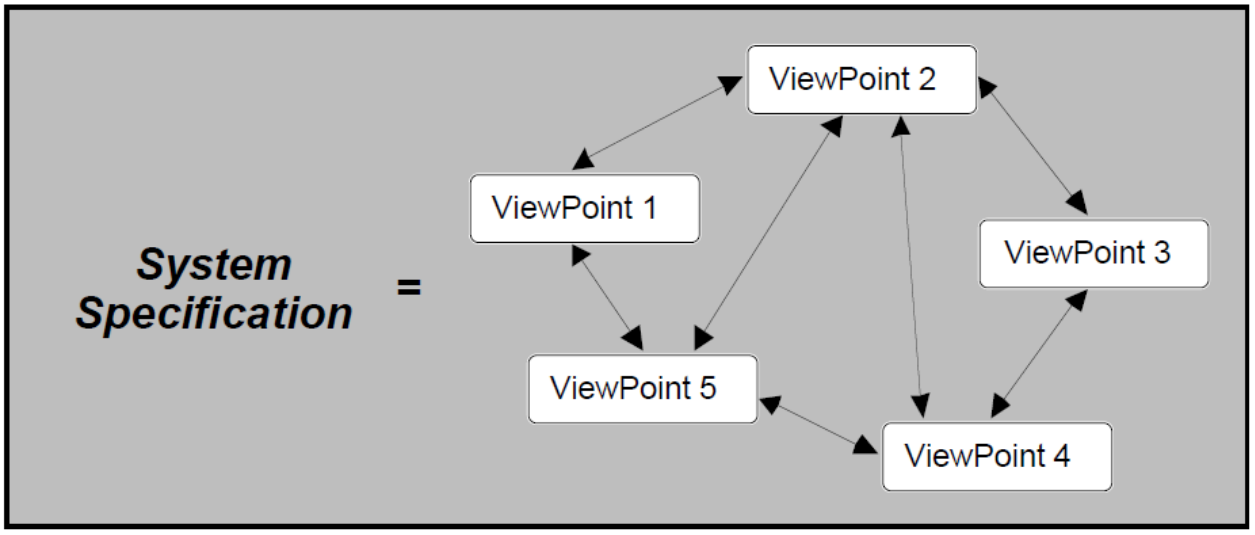


Рисунок 2.11 – Специфікація системи як структурована колекція ViewPoints. Стрілки представляють зв’язки

Таким чином, структура ViewPoints визнає цю неоднорідність продуктів розробки та надає засоби для їх структурування та організації. Все ще можливо, створити єдину ViewPoint, що представляє всю специфікацію системи, але вважається що таку «універсальну» ViewPoint складно побудувати, розуміти та підтримувати.

П’ять слотів ViewPoint інкапсулюють знання про представлення, процеси та специфікації, а власник ViewPoint додає знання домену. Хоча це охоплює різні види знань з розробки програмного забезпечення, необхідні для розробки системи, структура ViewPoints вимагає додаткової підтримки для взаємодії між ViewPoints. Підтримка такої взаємодії забезпечується інтерфейсом ViewPoint, який зберігає, перетворює та аналізує інформацію, що передається між ViewPoints. Інтерфейс ViewPoint визначає, які знання ViewPoint видимі для інших ViewPoints, приховуючи будь-які приватні або тимчасові знання в процесі. Основна роль інтерфейсу ViewPoint полягає в підтримці зв’язку між ViewPoint, який, у свою чергу, є засобом інтеграції методів.

Шаблони ViewPoint. Багато ViewPoints, хоч і містять описи різних доменів, можуть використовувати однакову схему представлення та стратегію розробки для створення цих описів. Причому, одна компанія може застосувати обмежену кількість технік або методів розробки в процесі, які вони постійно використовують для специфікації та побудови систем.

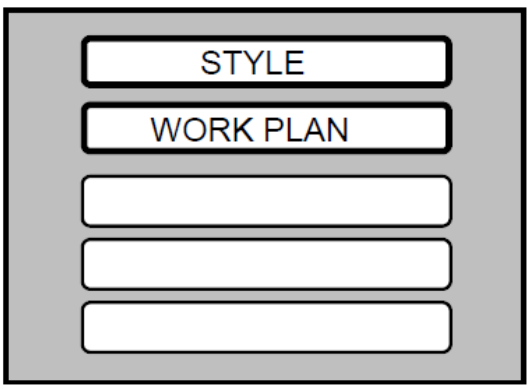


Рисунок 2.12 – Шаблон ViewPoint

Тому поняття «тип» або «клас» ViewPoint є корисним у цьому контексті. Такий тип ViewPoint називається «шаблоном ViewPoint». Шаблон – це ViewPoint, у якому розроблено лише слоти стилю та робочого плану (рис. 2.12).

Іншими словами, він містить лише визначення нотації та процесу розробки, розгорнутого ViewPoint. Таким чином, єдиний шаблон є описом єдиної техніки розробки.

ViewPoint створюється шляхом створення екземпляра шаблону ViewPoint. Один шаблон може бути створений багато разів, щоб отримати багато ViewPoints (рис. 2.13).

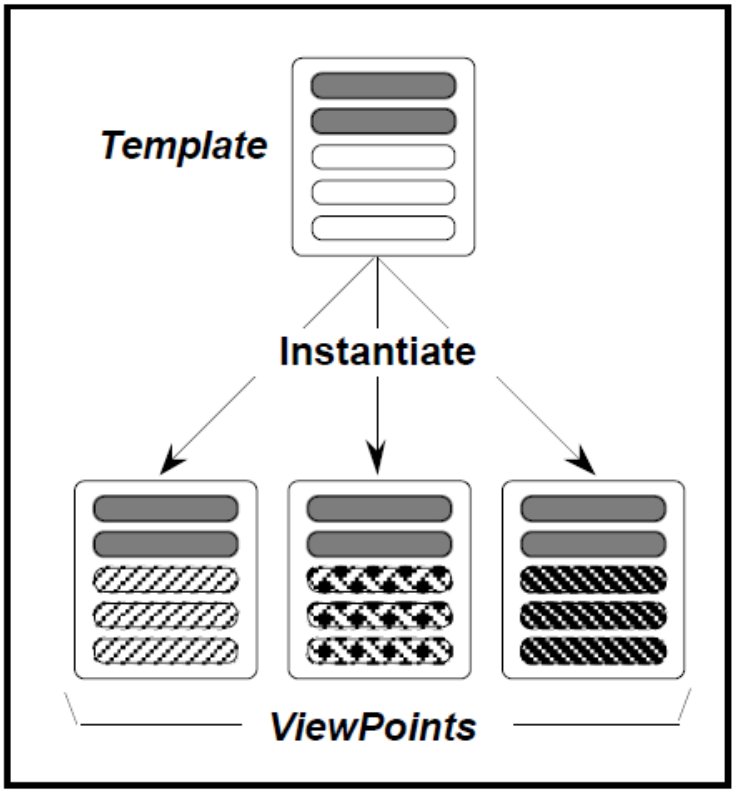


Рисунок 2.13 – Створення екземпляра шаблону Viewpoint

Крім того, один шаблон може бути розробкою, обмеженням або модифікацією іншого тобто один шаблон може успадкувати стиль або план роботи іншого. Таким чином, шаблон є багаторазовим описом, який полегшує повторне використання як шляхом інстанціювання, так і успадкування.

Методом розробки програмного забезпечення в контексті фреймворку ViewPoints є конфігурація (структурована колекція) шаблонів ViewPoint (рис. 2.14).

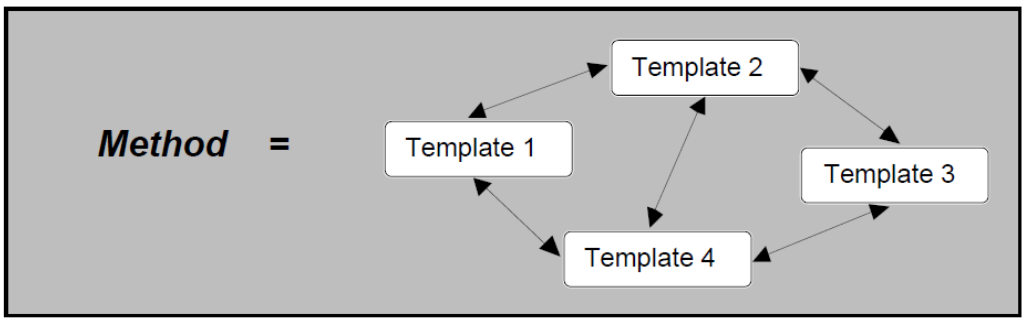


Рисунок 2.14 – Метод розробки програмного забезпечення в структурі ViewPoints

Іншими словами, метод - це структурований набір прийомів розробки. Кожну техніку, позначену шаблоном, можна розглядати як «примітивний», слабко пов’язаний, локально керований метод, з якого можуть бути створені (інтегровані) інші складені методи. Крім того, кожна методика може описувати окремий етап або крок у методі розробки, і в цьому випадку може знадобитися більш тісна інтеграція між шаблонами.

Висновки до розділу 2

В даному розділі досліджено сферу багатонапрямленої розробки програмного забезпечення та інтеграції методів. Розглянуто концепцію «відокремлення завдань» для зменшення складності розробки програмного забезпечення, зокрема, розглядається використання кількох ViewPoints як засобів для такого поділу. Також описано «інтеграцію» в контексті розробки програмного забезпечення, що базується на методах, з різними перспективами. Зокрема, основна увага приділяється інтеграції методів, яка визначена як ключова технологія для інших видів інтеграції, наприклад

інтеграції інструментів. Критично досліджено існуючі підходи, архітектури, техніки та механізми інтеграції методів. Також описано структуру ViewPoints для багатонапрямленої розробки програмного забезпечення. Визначено поняття «ViewPoint», різні види знань, яке вона інкапсулює, а також те, як ці знання розділені та структуровані на «слоти». Також представлено та визначено поняття «шаблон ViewPoint» та представлено як форму багаторазового примітивного методу. Усі поняття проілюстровано фрагментами прикладів. Далі розглядається структура, протиставляючи сильні сторони багатонапрямленого підходу до розвитку з труднощами досягнення інтеграції в таких умовах. Зокрема, у цьому розділі описано питання, пов’язані з підтримкою інструментів, моделюванням процесів, перевіркою узгодженості між точками перегляду та обробкою невідповідностей.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ МОДЕЛЕЙ ІНТЕГРАЦІЇ ТА АГРЕГАЦІЇ ПРОЦЕСІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Представлення архітектури та моделі об’єкта в інтеграційному фреймворці розробки

Архітектурна мета фреймворку ViewPoints полягає в тому, щоб відійти від централізованої архітектури до децентралізованої (рис. 3.1). Термін «архітектура» використовується для опису структурних елементів і комунікаційних принципів фреймворку. Таким чином, можна описувати централізовані архітектури, розподілені архітектури, конвеєрні архітектури тощо.

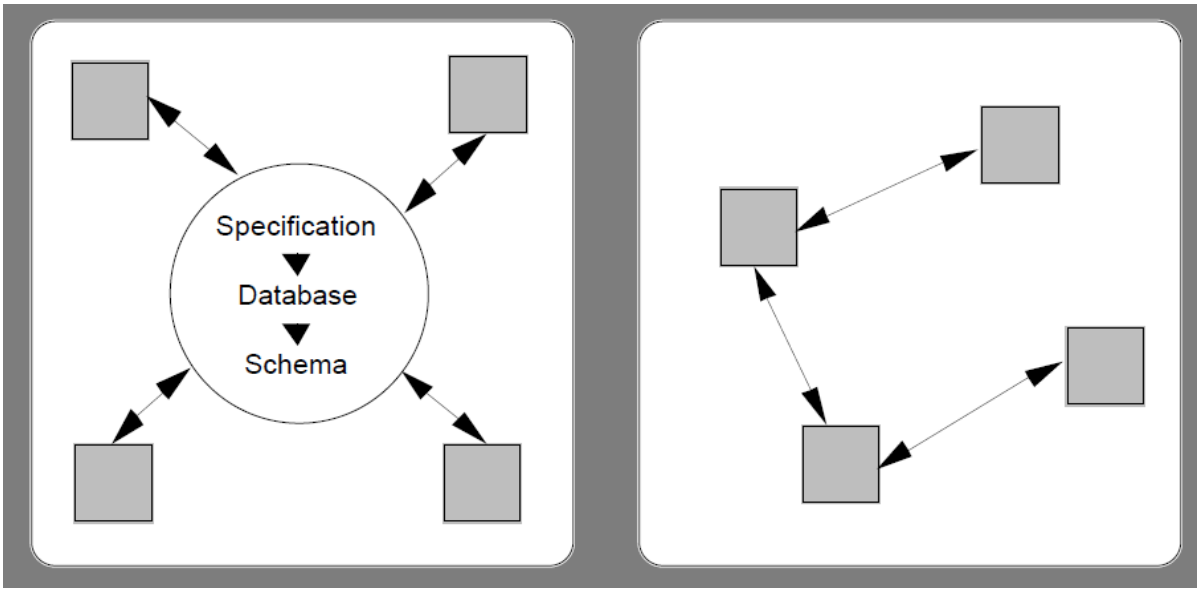


Рисунок 3.1 – Централізована та децентралізована архітектури

Таким чином, ViewPoints є дистрибутивним хоча це не виключає наявність централізованих ViewPoints, що містять глобальні дані або керуючу інформацію. Однак існує кілька рівнів децентралізації з різними рівнями складності їх досягнення. Наприклад, децентралізація специфікації шляхом її

розбиття на більш дрібні, більш керовані модулі є звичним явищем і точно вписується в модель розробки, орієнтовану на ViewPoint. З іншого боку, використання фізично або логічно децентралізованих баз даних для зберігання частин специфікації складніше. Більше того, якщо базові схеми специфікацій у різних базах даних також відрізняються, то керування процесом розробки специфікацій за сучасної технології є проблематичним.

Наш поточний підхід полягає в тому, щоб мати повністю децентралізовану архітектуру для фреймворку ViewPoints і працювати над невирішеними питаннями інтеграції та управління, а не приймати централізовану архітектуру, яку ми потім намагаємося децентралізувати. Ми також віддаємо перевагу використанню попарних перевірок зв'язків між точками огляду як засобу інтеграції.

Крім того, стверджується, що децентралізована архітектура фреймворку ViewPoints є кращим відображенням «реального» розвитку кількома учасниками розробки, які мають різні точки зору. Учасники розробки природно розподілені, і хоча вони можуть ділитися знаннями або навіть мати спільний погляд на світ, ці знання розподіляються між ними і не проводиться централізовано. Насправді, як правило, багато надлишкової або дубльованої інформації зберігається різними учасниками розробки, і тому єдиний, централізований масив інформації є неточним представленням процесу.

Підтримка інструментів. Сфера застосування інструментальної підтримки VOSE зосереджена навколо двох видів діяльності: «розробка методів» і «використання методів».

Для підтримки розробки методів потрібні інструменти для опису та складання шаблонів, а також для створення інструментів CASE для підтримки розробки специфікації ViewPoint на основі цих шаблонів. Для опису шаблону потрібні такі інструменти, як текстові та графічні редактори для визначення стилів шаблону, а також інструменти моделювання процесу для визначення робочих планів шаблону. Бібліотеки шаблонів також корисні

для повторного використання попередньо визначених шаблонів. Створення інструментів CASE вимагає або програмування або наборів мета CASE-інструментів для створення або генерації, відповідно, інструментів для підтримки різних шаблонів.

Для підтримки використання методу, тобто розробки програмного забезпечення, орієнтованого на ViewPoint, потрібна інфраструктура розподіленої системи для підтримки розподілених точок огляду та їх взаємодії, а також гнучкого середовища, у якому можна розробляти точки огляду та керувати ними. Останній включає середовища, які полегшують використання інструментів, створених або згенерованих процесом розробки методів.

Структура ViewPoints також є засобом інтеграції інструментів. Це досягається шляхом розгляду інтеграції інструментів як окремого випадку інтеграції методів. Інтеграція методів, як ми обговоримо в розділі 6, є процесом інтеграції шаблонів ViewPoint за допомогою перевірок між ViewPoint, керованих моделлю процесу.

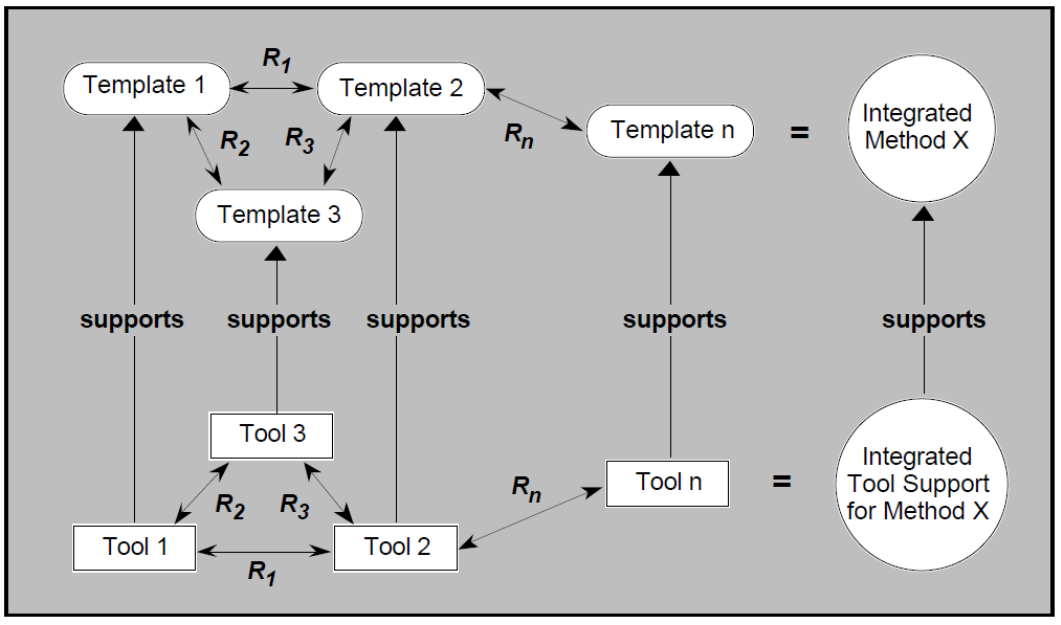


Рисунок 3.2 – Інструментальна інтеграція як випадок інтеграції методу

Якщо визначення шаблонів ViewPoint зроблено достатньо точними та повними, щоб генерувати CASE-інструменти з цих визначень, тоді ті самі перевірки та пов’язана модель процесу, яка забезпечила інтеграцію методів, також можна використовувати для інтеграції інструментів (рис. 3.2).

Модель об'єкта. Фреймворк ViewPoints — це об’єктний фреймворк, а View Points аналогічні «об’єктам» у традиційній об’єктній моделі. Вони мають «стан» (із слотами специфікації та робочих записів як «змінними екземплярів») і «поведінку» (з «методами», визначеними в робочому плані і слотами стилю). Вони мають унікальну ідентифікацію, позначену доменом і власником і також створені з «класів» (шаблонів), але наразі немає явної підтримки успадкування.

Таблиця 3.1

Концепція VOSE і парадигма ООП

Object-Orientation	VOSE
Object (instance)	ViewPoint
Class	Template
Encapsulation of state and behaviour	Encapsulation of state in specification and work record
Object identity	ViewPoint domain and owner
Information hiding: objects can only be changed by object operations	Information hiding: ViewPoint specifications can only be changed by work plan actions
Inheritance	Not available (omitted to maximise encapsulation of templates and distribution of ViewPoints)
Polymorphism: the binding of a message to different methods, depending on the class of the receiving object	Polymorphism “simulated” (because we have neither inheritance nor dynamic binding): identical work plan actions may be used to build specifications based on different styles, depending on the template instantiated
Message passing	Message passing - available but not in the strict object-oriented sense; rather, in the distributed systems context: via inter-ViewPoint communication protocol

Успадкування не є частиною поточної структури з прагматичних міркувань, оскільки існує певні протиріччя між упадкуванням та інкапсуляцією, яка необхідна для ефективного розподілу. Таким чином, хоча концептуально успадкування між шаблонами є бажаним, сучасна технологія ускладнює ефективну реалізацію в розподіленому середовищі. Таблиця 3.1 підсумовує ці та інші аналогії між об'єктно-орієнтованим і VOSE.

Методи інтеграції. Основною метою є досягнення інтеграції методів у контексті багато напрямленої розробки програмного забезпечення, на прикладі VOSE. Фреймворк ViewPoints полегшує розробку програмного забезпечення шляхом розгортання кількох ViewPoints для представлення кількох точок зору та учасників розробки. Архітектура фреймворку за своєю суттю є децентралізованою, що дозволяє розподілену одночасну розробку. Шаблони ViewPoint полегшують визначення методів розробки, на основі яких створюються та розвиваються ViewPoints. Один шаблон ViewPoint представляє примітивний метод, тоді як набір шаблонів становить більш складний складений метод.

Таким чином, структура досягає поділу інтересів між ViewPoints з точки зору і учасників розробки і різних видів інженерних знань програмного забезпечення, які інкапсулює кожна ViewPoint. Розділення проблем також досягається на рівні розробки методу в термінах різних шаблонів, які становлять методи, і на рівні використання методу в термінах різних точок огляду, які утворюють специфікації системи.

Однак інтеграція цих проблем на різних етапах не є такою простою та вимагає зміни структури. Це, у свою чергу, впливає на характер процесу розробки програмного забезпечення, орієнтованого на ViewPoint, і спосіб, у який виконуються дії в межах структури.

На рис. 3.3 показано можливості для інтеграції в структуру ViewPoints. Це також ілюструє необхідність вивчення інших видів діяльності для досягнення бажаної інтеграції методів.

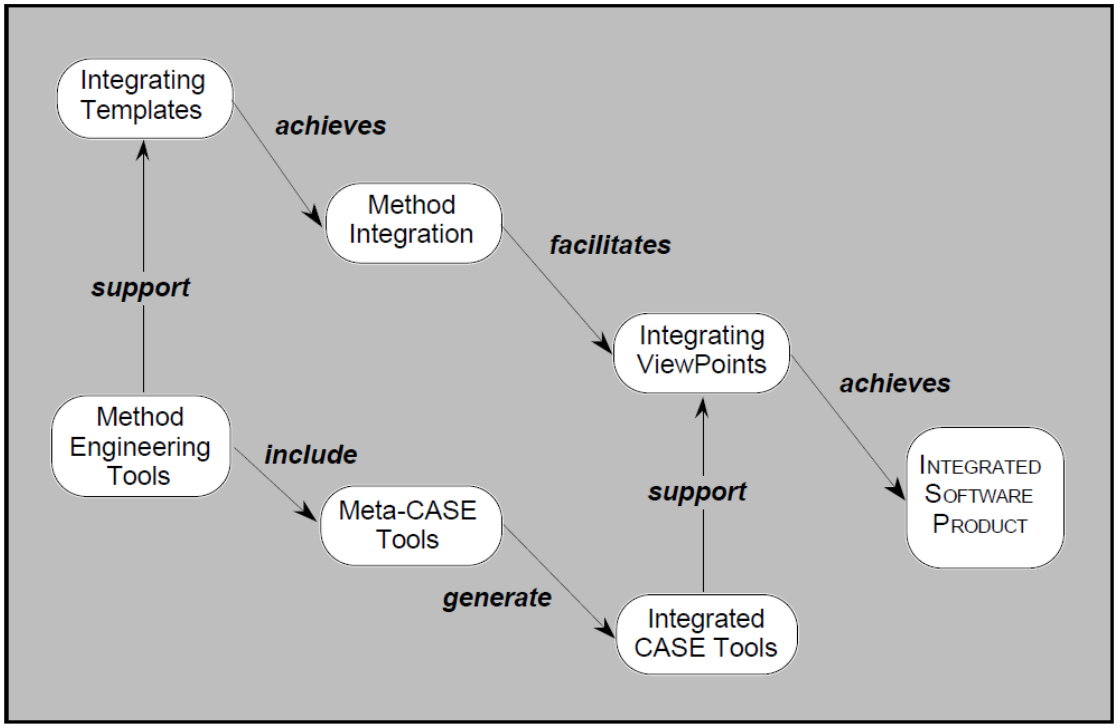


Рисунок 3.3 – Можливості інтеграції ViewPoints framework

Під час розробки методів інтеграція відбувається на рівні шаблону, тобто гарантується, що зв'язки між шаблонами визначені та виражені. Під час використання методу (розробка ViewPoint) інтеграція відбувається між ViewPoints, тобто гарантується, що правила між ViewPoint викликаються у відповідний час і застосовуються правильно.

Розробка методів у структурі ViewPoints спрощується тим фактом, що єдиний метод розробки програмного забезпечення може бути розроблений і створений модульним способом шляхом як визначення окремих шаблонів, так і повторного використання попередньо визначених шаблонів із бібліотеки повторного використання таких шаблонів. Підтримка інструментів для цього процесу також спрощена тим, що інструменти розробки методів у цьому контексті є або структурованими або мета-CASE-інструментами для генерації CASE-інструментів для підтримки, переважно простих, шаблонів.

Інтеграція методів у структуру ViewPoints зрештою пов'язана зі зв'язками між ViewPoint – як вони виражаються, коли викликаються та як застосовуються. Ці зв'язки спочатку проявляються як зв'язки всередині та

між шаблонами, які називають зв’язками між ViewPoints. Викладено вимоги інтеграції методів під час розробки методів, коли визначено шаблони і «вимірюється» збереження зв’язків між точками огляду, створеними з цих шаблонів.

3.2 Розробка методу на основі концепції ViewPoints

Багатонапрямлена розробка програмного забезпечення залучає кількох учасників, які незмінно дотримуються різних поглядів на проблему чи область вирішення. Ці учасники включають клієнтів, які мають різні суперечливі та взаємодоповнюючі вимоги до програмної системи, яку вони бажають придбати і розробники, які повинні виявити, визначити, проаналізувати та підтвердити ці вимоги, а потім спроектувати та створити програмну систему, яка задовольняє ці вимоги.

Іншим учасником процесу розробки програмного забезпечення є інженер з документування вимог (Method Engineer). Загалом, розробник методів відповідає за проектування та конструювання методу розробки, який, у свою чергу, надає розробникам систематичні процедури та вказівки для розгортання однієї або кількох нотацій для опису проблемних областей. Дедалі частіше роль інженера стає більш специфічною для сфери застосування, з «налаштованими» або «специфічними» методами, зібраними для конкретних організацій або проектів. Цьому сприяють інструменти комп’ютерної розробки методів (CAME), наприклад мета-CASE-інструменти.

Теоретично діяльність інженерів з документування вимог передує діяльності розробників програмного забезпечення. На практиці ця роль і ролі розробників програмного забезпечення та клієнтів нерозривно пов’язані та часто збігаються (рис. 3.4). І клієнти і розробники програмного забезпечення висуватимуть вимоги до розробника методів: клієнт може вимагати, щоб розробник програмного забезпечення використовував певний метод, а системні вимоги можуть вимагати налаштування цього методу.

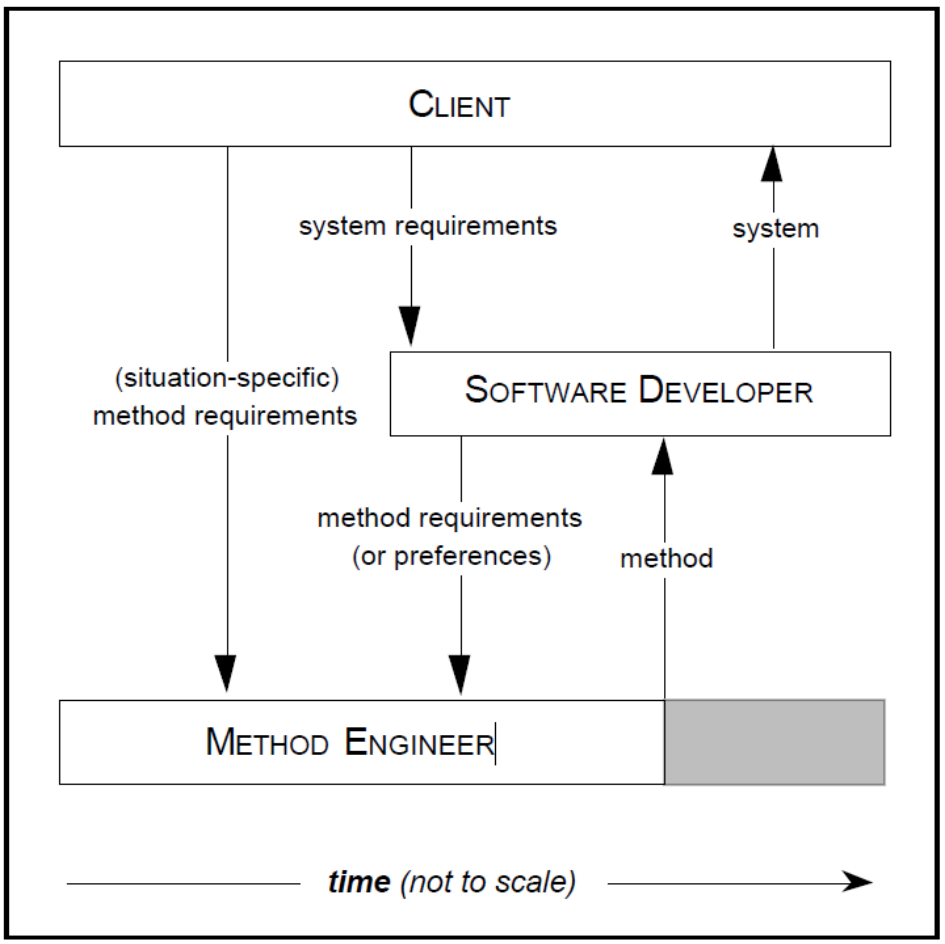


Рисунок 3.4 – Учасники розробки програмного забезпечення

На рисунку 3.4 заштрихована частина діяльності розробника методу вказує на те, що його участь у розробці може тривати навіть після надання методу розробки програмного забезпечення розробнику програмного забезпечення.

В структурі ViewPoints методом розробки програмного забезпечення є конфігурація (структурована колекція) шаблонів ViewPoint. Таким чином, ці шаблони є основними фрагментами (блоками) методів, з яких створюються специфікації системи (конфігурації ViewPoint). Структура методу в цьому контексті визначається внутрішніми та міжшаблонними зв'язками. Внутрішньошаблонні зв'язки — це зв'язки між ViewPoints, створеними з одного шаблону, тоді як зв'язки між шаблонами — це зв'язки між ViewPoints, створеними з різних шаблонів.

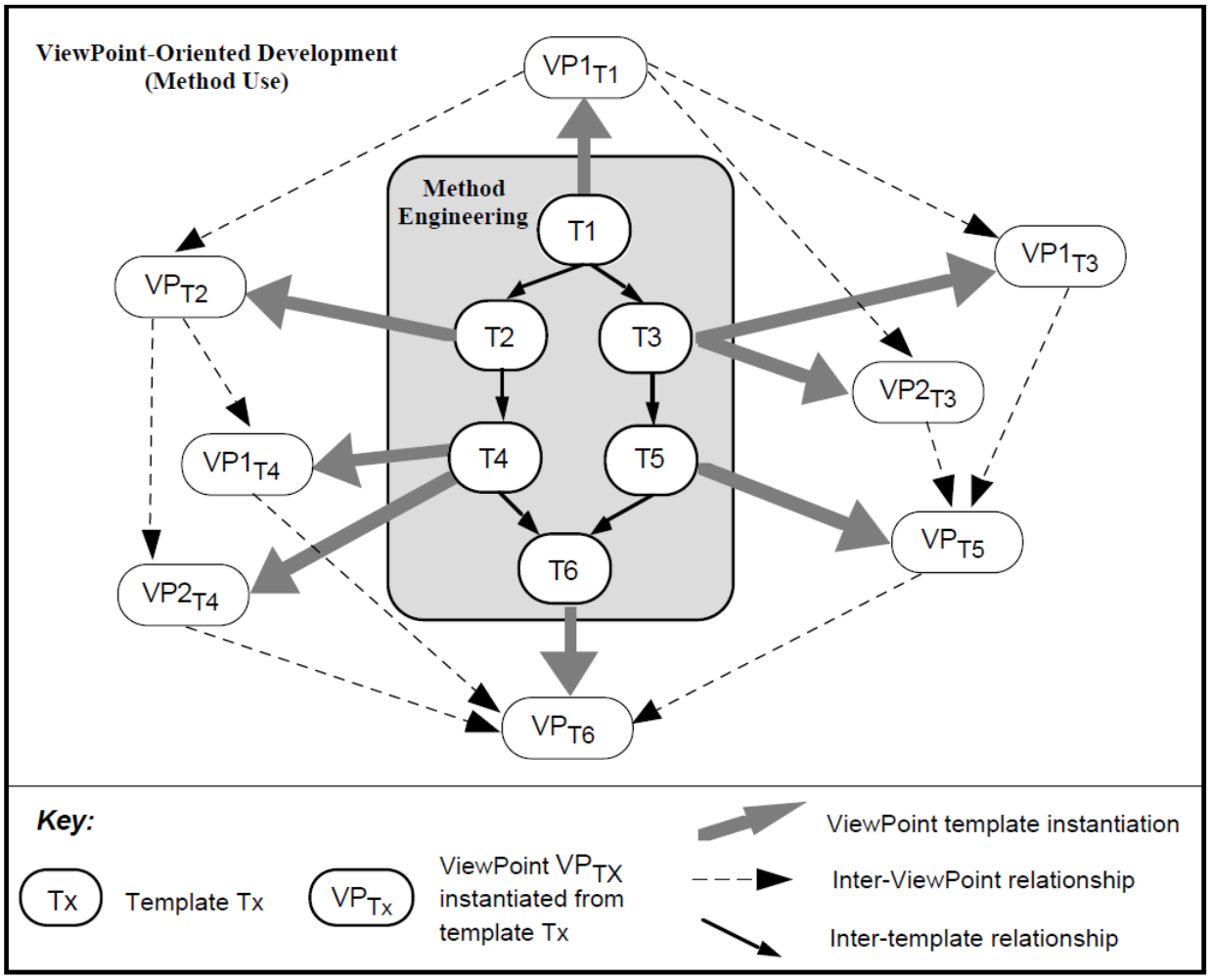


Рисунок 3.5 – Розробка методу з використанням концепції ViewPoint

На рисунку 3.5 зображено ряд дій, які відбуваються як під час розробки методу, так і під час використання методу. Конкретний метод, показаний на рисунку, складається з шести шаблонів, позначених від T1 до T6. Стрілки між шаблонами позначають зв’язки між ними. Таким чином, T1 пов’язаний як з T2, так і з T3, тоді як T2 і T3 пов’язані з T4 і T5 відповідно. І T4, і T5 пов'язані з T6. Напрямки стрілок також вказують на «бажаний» або рекомендований порядок діяльності з розробки високого рівня, наприклад, порядок, у якому повинні створюватися екземпляри різних шаблонів. У цьому контексті роль інженера полягає в тому, щоб ідентифікувати, описати та зв’язати різні шаблони, які складають необхідний метод.

Товсті сірі стрілки на рисунку 3.5 представляють дії створення екземплярів у шаблонах («Дії тригера ViewPoint», описані в попередньому розділі). Результатом дії створення екземпляра на шаблоні Tx є ViewPoint VP Tx. Треба зауважити, що один шаблон, такий як T4, може бути створений багато разів, щоб отримати стільки ViewPoint (тому, наприклад, T4 створено двічі на рисунку, щоб отримати ViewPoint VP1 T4 і VP2 T4).

Нарешті, різні ViewPoint, створені в результаті цього процесу побудови екземплярів шаблонів, самі пов’язані екземплярами зв’язків, визначених між шаблонами, з яких вони були створені. Таким чином, зв’язок між шаблонами T2 і T4, стає зв’язком між ViewPoints між усіма ViewPoints, створеними з T2 і T4 відповідно. Іншими словами, VPT 2 пов’язаний як з VP1 T 4, так і з VP2 T4 через екземпляри зв’язку. Створення екземплярів шаблонів та їхніх зв’язків — це діяльність, яка відбувається під час так званої «розробки, орієнтованої на ViewPoint», тобто під час використання методу.

Рисунок 3.5 також ілюструє деякі складності методів розробки програмного забезпечення та різні способи, якими вони можуть бути розгорнуті. Загалом, методи не передбачають послідовний ряд процедур, а складаються з кількох слабко пов’язаних етапів, кожен з яких містить одну або більше методик або підтехнологій. Таким чином, розглядаючи шаблони ViewPoint як компоненти в конфігурації, специфікація системи потім розробляється шляхом створення екземплярів цих шаблонів, які створюються або «породжуються» динамічно в ході розробки.

На рисунку 3.5 не показано ітеративний процес уточнення, за допомогою якого окремі ViewPoints (частково) розробляються, а потім модифікуються та еволюціонують у результаті обходу стрілок взаємозв’язку між ними. Для того, щоб зрозуміти та слідувати такому процесу розробки, орієнтованому на ViewPoint, необхідно також зрозуміти процес розробки методів. Таким чином, метод у структурі ViewPoints повинен бути розроблений і створений таким чином, щоб дозволяти створювати ViewPoints

за потреби, частково розроблятися, якщо це буде необхідно, і періодично перевірятися в процесі розробки.

3.3 Проектування методу на основі концепції ViewPoint

Метою розробки методів у структурі ViewPoints є створення конфігурації шаблонів ViewPoint. Іншими словами, роль розробника методу полягає в тому, щоб визначити, спроектувати та створити шаблони, які складають метод. Як правило, розробники методів або «винаходять» нові методи з нуля, або збирають (повторно використовують і адаптують) методи з бібліотеки повторного використання фрагментів методів (шаблонів). Спосіб, у який методи «розрізаються» на їх складові шаблони є центральною діяльністю з розробки методів, яка спирається на досвід і застосування деяких простих евристик. На рисунку 3.6 показано процес розробки методу, орієнтованого на ViewPoint, який більш детально описано нижче.

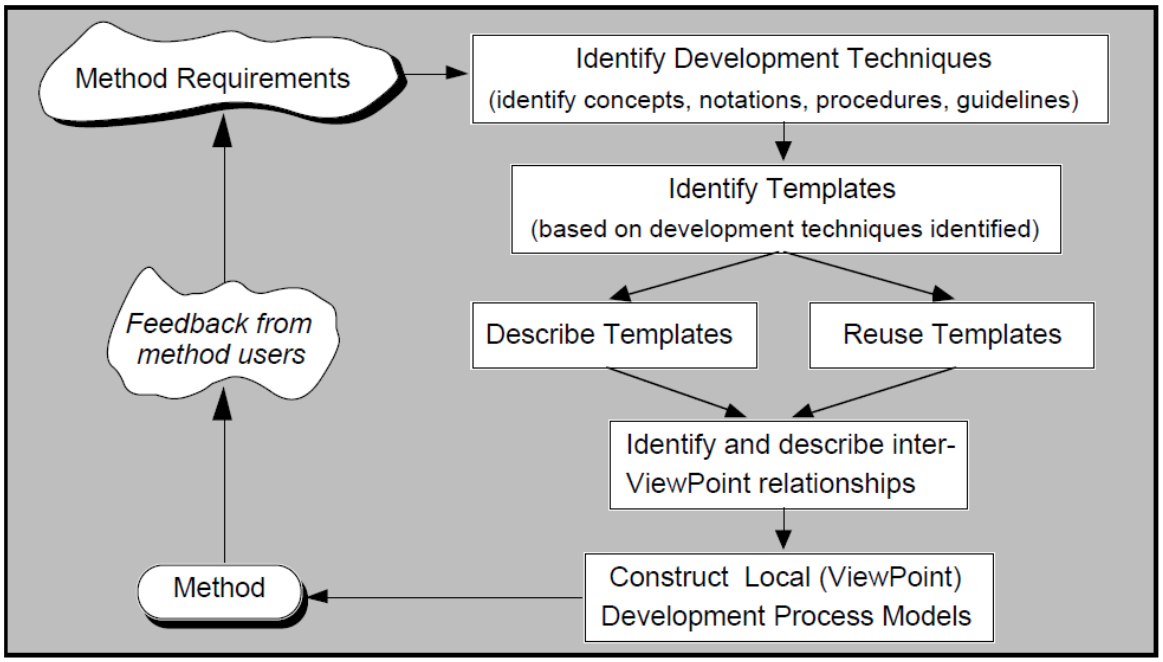


Рисунок 3.6 – Розробка та побудова методу на основі ViewPoints framework

Виходячи з невизначених вимог до методу розробки програмного забезпечення, інженер визначає методи розробки, які використовуватиме цільовий метод. Це включає в себе визначення різних нотацій і стратегій розвитку які будуть частиною цього методу.

Наприклад, згідно Object Modelling Technique (OMT) було визначено три типи технік моделювання, необхідних для створення об’єктно-орієнтованого дизайну, а саме об’єктне, динамічне та функціональне моделювання. З цією метою адаптовано три відомі методи розробки або побудови діаграм (зв’язок сутності, перехід стану та потік даних) і використано їх (відповідно) для представлення трьох моделей OMT.

Наступним кроком є визначення шаблонів ViewPoint, які необхідно створити, щоб описати методи, визначені вище. Гарною евристикою в таких ситуаціях є ідентифікація якомога більшої кількості простих технік розробки (наприклад, тих, які використовують прості нотації), а потім вибір єдиного шаблону для опису кожної техніки. Звичайно, інженер не буде вибирати які методи розробки і шаблони використовувати, а буде обмежений щодо видів технік або нотацій, умовами клієнтів.

Після того, як необхідні шаблони визначено, їх потрібно визначити більш точно. Іншими словами, стиль і слоти робочого плану для кожного шаблону повинні бути розроблені. Як альтернатива, шаблони, вже описані в інших методах, можуть бути адаптовані для задоволення вимог нового методу. Така адаптація зазвичай включає зміни в локальному стилі шаблону та діях робочого плану шаблону.

Дії перевірки Inter-ViewPoint застосовують правила між ViewPoint. Ці правила є специфічними для методу атрибутами шаблонів. Вони описують внутрішні та міжшаблонні зв’язки та є єдиними нелокальними властивостями шаблонів, якими розробникам методів потрібно займатися на цьому етапі. Якщо шаблон повторно використовувався з іншого контексту (методу), ці правила також потрібно змінити або повністю переписати. Якщо шаблон було визначено з нуля, ці правила потрібно писати з нуля. Правила Inter-ViewPoint

відіграють низку важливих ролей у розробці та використанні методів, і вони будуть обговорюватися більш детально в розділі 6. Однак насамперед вони пов’язують між собою різні шаблони методів і, отже, є засобом інтеграції методів. Інтеграція методів сприяє багатомірній розробці програмного забезпечення інтегрованих програмних систем.

Останнім етапом розробки методу є визначення локальної моделі розробки (процесу) для кожного окремого шаблону. Кожна модель процесу визначає, коли мають відбуватися різні дії робочого плану; наприклад, коли потрібно викликати та перевірити правила між точками перегляду. Таким чином, локальні моделі процесів ViewPoints також служать для синхронізації зв’язку між ViewPoints шляхом координації виклику та застосування дій перевірки між ViewPoint. Координацію моделей процесу ViewPoint у цьому параметрі також можна назвати інтеграцією процесу.

Результат процесу розробки методу, описаного вище, є методом , який задовольняє початкові вимоги та може бути наданий розробникам програмного забезпечення та використаний ним для розробки відповідних систем програмного забезпечення. Хоча динамічна еволюція методів (під час використання методів) наразі не підтримується фреймворком (якщо створено екземпляри складових шаблонів цих методів), очевидно, що це проблема, яку потрібно вирішити, а потім ретельно підтримувати та контролювати.

Структура ViewPoints надає розробникам інфраструктуру для повторного використання. Найбільш очевидним багаторазовим компонентом у структурі є шаблон ViewPoint, який можна повторно використовувати принаймні трьома різними способами (схематично показано на рисунку 3.7):

• Перший спосіб — розглядати шаблони як окремі компоненти, що належать до «бібліотеки повторного використання» (шаблонів для повторного використання). Їх можна окремо «підключити» до відповідного методу, зазвичай після певної адаптації. Повторне використання шаблонів у такий спосіб може заощадити значний час і зусилля розробників методів під час проектування та побудови методу, зменшивши необхідність визначати

шаблони з нуля. Однак проблеми класифікації, підтримки та доступу до великих бібліотек багаторазових компонентів залишаються. Крім того, розробникам методів необхідно докласти зусиль у плануванні та проектуванні, щоб створити та заповнити такі багаторазові бібліотеки шаблонів.

• Другий спосіб полягає в повторному використанні шаблонів ViewPoint за допомогою техніки інженірингу. Шаблони аналогічні «типам» або «класам», що містять загальну інформацію, яка передається екземплярам цих шаблонів, що дозволяє кільком ViewPoints використовувати однакові нотації або стратегії розробки, якщо вони створені з одного шаблону. Проте повторне використання шляхом інстанціювання зазвичай означає, що інформація «рівня типу» (рівня шаблону) дублюється в кожному екземплярі.

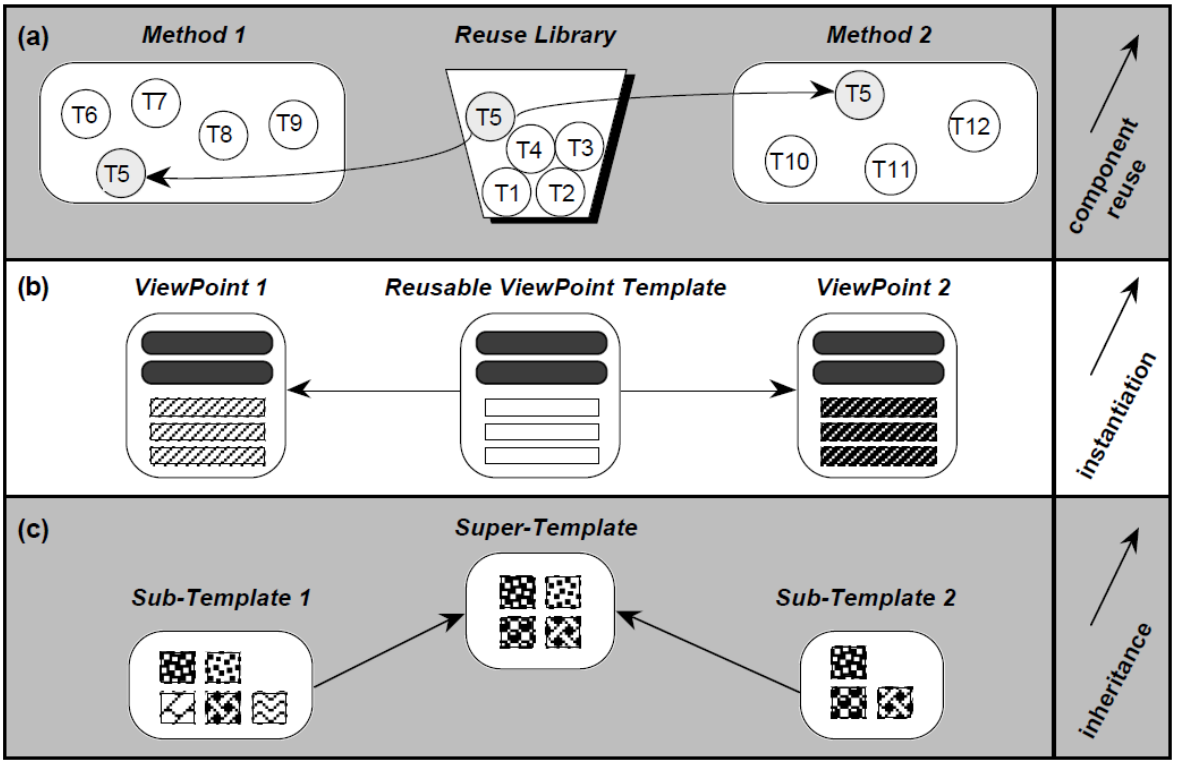


Рисунок 3.7 – Шаблони ViewPoint як засоби для повторного використання. a) шаблони як багаторазові компоненти; b) Кілька ViewPoints, створених з одного (багаторазового) шаблону; c) Шаблони, що обмінюються (повторно використовують) знання через успадкування.

• Третій спосіб, який не повністю підтримується, полягає у повторному використанні знань шаблонів через успадкування. Багато шаблонів можуть мати однакове представлення або знання процесу. Щоб уникнути дублювання таких знань у кожному новому створеному шаблоні, можна розробити структуру «супершаблон/підшаблон» (суперкласи / підкласи в структурах об'єктно-орієнтованого програмування). Наприклад, можна визначити «абстрактний шаблон» (абстрактний клас), який описує загальну техніку діаграми потоку даних з двома додатковими під шаблонами для відповідного представлення двох нотаційних варіацій цієї техніки.

Повторне використання у структурі ViewPoints є допоміжним засобом для побудови методів. Можна заощадити значний час і зусилля, повторно використовуючи існуючі шаблони та адаптуючи їх до вимог нових методів. Шаблони є компонентами, які можна багаторазово використовувати, за винятком тієї частини їхніх робочих планів, яка визначає зв'язки між ViewPoint. Строго кажучи, дії перевірки між ViewPoint слід визначати окремо від окремих шаблонів, оскільки вони не є частиною жодної окремої методики розробки як такої, а радше описують зв'язки, які інженери методів хочуть виразити між цими техніками. Вони також більшою мірою залежать від методів і тому менш придатні для повторного використання, ніж решта елементів шаблонів ViewPoint.

Варто зауважити, що з точки зору користувача методу, структура ViewPoints також полегшує повторне використання. Самі ViewPoints є повторно використовуваними (спеціальними) компонентами, які можна класифікувати, абстрагувати та комбінувати.

Загалом використання методу – це розгортання його для визначення та розробки системи програмного забезпечення. У структурі ViewPoints це характеризується створенням і розвитком багатьох ViewPoints, кожна з яких стосується певного аспекту або частини системи, що розробляється. Таким чином, розробка програмного забезпечення, орієнтованого на ViewPoint,

підтримує поширення кількох представлень під час вимог, кількох рішень під час проектування та мультипарадигмального програмування під час впровадження.

3.4 Імплементація методів із застосуванням правил

Під час проектування методу ідентифікуються складові шаблони методу та визначаються зв’язки, які повинні зберігатися між ViewPoints, створеними з цих шаблонів. Таким чином, правила між ViewPoints в цьому параметрі представляють «гіпотетичні» зв’язки між точками огляду, які ще не створені. Це пояснюється тим, що розробка методів відбувається на рівні «типу» (шаблону), а не на рівні «примірника» (точка перегляду). Іншими словами, розробник, який виражає зв’язки між вихідними та кінцевими ViewPoints, VP_S та VP_D відповідно, фактично говорить: якщо ViewPoints VP_S та VP_D існують, то між ними має існувати зв’язок. Пунктирні лінії на рисунку 3.7 позначають ViewPoints, які ще не створені, і, отже, зв’язок який ще не підтримується.

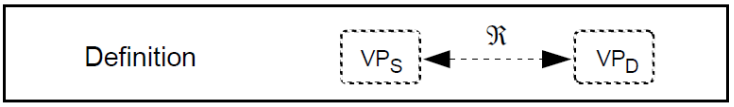


Рисунок 3.7 – Визначення правила Inter-ViewPoint

Таким чином, визначення правила між ViewPoints – це процес вираження взаємозв’язків між шаблонами та між Inter-шаблонами – інтеграція методів основних елементів.

Під час розгортання методу розробники програмного забезпечення розробляють окремі ViewPoints. Кожна окрема модель процесу ViewPoint визначає точки, у яких правила між ViewPoint повинні бути викликані, а потім перевірені. Виклик правила між точками огляду з будь-якої окремої

ViewPoints, VP_S , еквівалентно наступному: якщо існує кінцева ViewPoints, VP_D тоді зв'язок має зберігатися між VP_S і VP_D . Суцільна лінія навколо VP_S на рисунку 3.8 вказує на те, що VP_S уже існує (це ViewPoints, з якої було викликано правило), тоді як переривчасті лінії вказують на те, що VP_D та відношення ще не створено та не підтримується, відповідно. Звичайно, правило також говорить про те, що перед тим, як ми перейдемо до наступного етапу (застосування правила між ViewPoints), необхідно створити одну або кілька цільових ViewPoints. Це виконується за допомогою дії тригера ViewPoint.

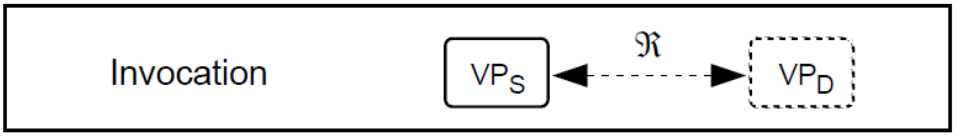


Рисунок 3.8 – Виклик правила Inter-ViewPoint

Таким чином, виклик правила між ViewPoints — це процес визначення точок, у яких необхідно перевірити узгодженість (інтеграцію).

Після ідентифікації двох пов'язаних точок VP_S і VP_D, необхідно перевірити правило між точками які їх пов'язує. Таким чином, застосування правила Inter-ViewPoint є процесом запитування: чи зберігається визначений зв'язок між VP_S і VP_D ? Пунктирна лінія на рисунку 3.9 представляє співвідношення яке ще потрібно перевірити.

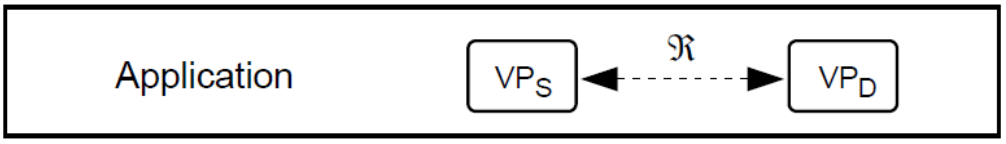


Рисунок 3.9 – Застосування правила Inter-ViewPoint

Якщо виявлено, що зв'язок не виконується, тоді можна запустити процес обробки невідповідності, кінцевою метою якого є збереження цього

зв’язку. Мета всього процесу керування узгодженістю, представленого цією інтеграційною моделлю ViewPoints, полягає в тому, щоб досягти узгодженості між точками шляхом збереження відповідних правил між ними. Таким чином, незалежно від того, чи виконуються правила між точками відразу чи після певної обробки неузгодженості, остаточний бажаний стан показано на рисунку 3.10, на якому підтверджено, що співвідношення зв’язку між VP_S і VP_D, а тому суцільні лінії в усіх частинах діаграми.

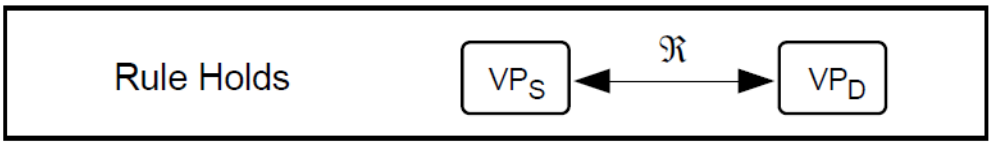


Рисунок 3.10 – Підтвердження дії правила Inter-ViewPoint

Таким чином, застосування правил між ViewPoints — це процес перевірки узгодженості між двома взаємодіючими точками з метою зробити зв’язок виражений застосованим постійним правилом.

Правила Inter-ViewPoint розкривають різну структурну інформацію про методи, процеси розробки та специфікації системи залежно від етапу, на якому вони спостерігаються. Ця інформація може бути корисною при побудові інструментальної підтримки для фреймворку ViewPoint і може використовуватися для візуалізації структури методів, стану процесів розробки та структури систем, що розробляються.

Під час проектування методів, правила між точками описують зв’язки між шаблонами та між ними. Таким чином, у поєднанні з шаблонами, які вони стосуються, ці правила надають інформацію про структуру методів, які вони інтегрують. Метод у структурі ViewPoints — це конфігурація (структурована колекція) шаблонів ViewPoint. Ця структура визначається зв'язками (вираженими як правила) між складовими шаблонами методу. Ці відношення позначені стрілками на рисунку 3.11.

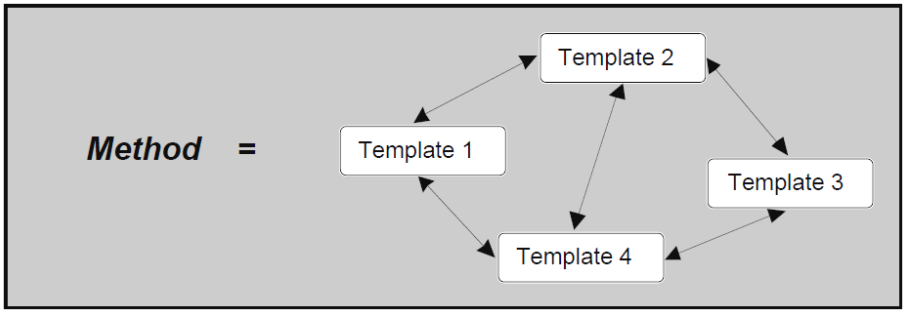


Рисунок 3.11 – Структура методу, що базується на ViewPoint і визначається правилами між шаблонами

В пропонованій моделі інтеграції ViewPoints визначення правил між ViewPoint також є кроком, на якому визначається структура методів.

У будь-який момент процесу розробки не обов'язково будуть створені всі необхідні ViewPoints. Крім того, у будь-який момент під час розробки кілька точок можна створити безпосередньо з поточної конфігурації. Які точки мають бути створені, визначається правилами між ними, які були визначені під час розробки методу. Ці правила представляють зв'язки між ViewPoints в конфігурації, і тому їх можна використовувати для визначення та «підказки» відсутніх точок. Рисунок 3.12 є «моментальним знімком» процесу розробки, орієнтованого на ViewPoint. Пунктирні лінії представляють ViewPoints, які ще не створені, але які «досяжні» з існуючих точок (вони досягаються шляхом виклику правил між точками показаних на рисунку).

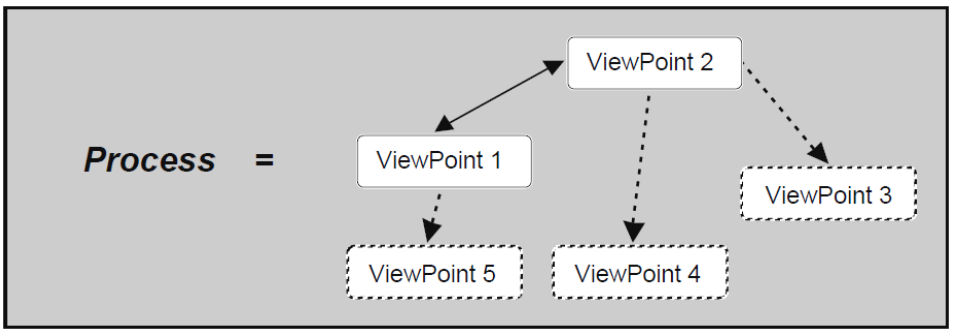


Рисунок 3.12 – Процес розробки визначається правилами між ViewPoint. Точки 1 і 2 були створені, а 3, 4 і 5 можуть бути побудовані безпосередньо з них

Метою процесу розробки, орієнтованого на ViewPoint, є створення системної специфікації (або системи програмного забезпечення), яка є узгодженою (відповідно до деяких правил між ViewPoint). У структурі ViewPoints метою є створення інтегрованої конфігурації ViewPoints, у якій зв'язки, які були визначені під час проектування методу, були перевірені та знайдені (або створені) для збереження. Рисунок 3.13 ілюструє, як структура специфікації системи визначається правилами між точками. Пунктирні лінії вказують на взаємозв'язки між ViewPoint, які ще потрібно перевірити та які не обов'язково дотримуються.

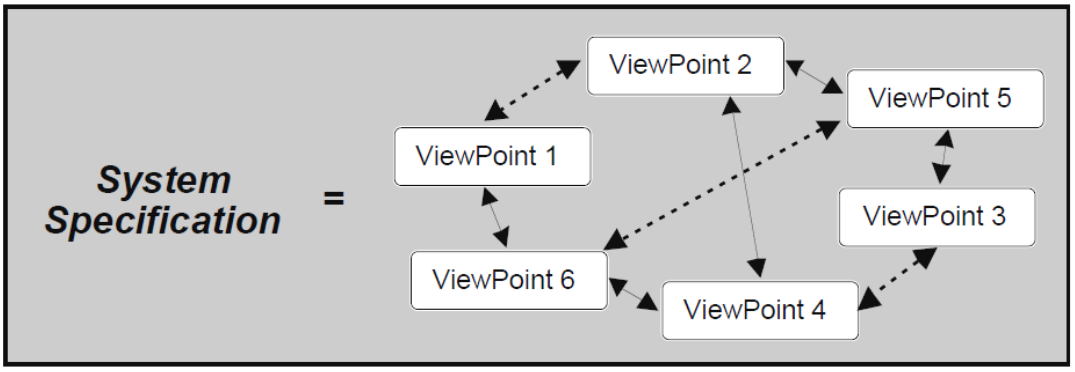


Рисунок 3.13 – Специфікація системи, що визначається правилами між ViewPoint

Рисунок 3.13. також підкреслює потенційну проблему масштабу, яку створює така конфігурація ViewPoints.

У моделі інтеграції ViewPoints застосування правил між точками є етапом на якому перевіряються зв'язки між точками огляду. Структурована колекція (конфігурація) точок, створена наприкінці цього кроку, є системною специфікацією або програмним забезпеченням.

Щоб ефективно підтримувати багатонапрямлену розробку програмного забезпечення, така розробка має базуватися на інтегрованих методах розробки програмного забезпечення. Таким чином, у контексті структури ViewPoints інтеграція методів є засобом для досягнення інтеграції ViewPoints.

Інтеграція ViewPoints — це створення узгодженої колекції ViewPoints шляхом застосування правил узгодженості між ViewPoints. Ці правила визначають попарні зв’язки між точками огляду та є ключем до структурування та інтеграції методів, процесів і специфікацій у цьому параметрі.

Висновки до розділу 3

Отже, в цьому розділі представлено структуру ViewPoints для розподіленої багатонапрямленої розробки програмного забезпечення. Було наведено структурні елементи фреймворку, ViewPoints і шаблони ViewPoint, а також було представлено можливості підтримки ViewPoint-Oriented Software Engineering (VOSE). Представлено розробку методів у структурі ViewPoints і досліджено як метод розробляється та створюється розробниками і подається як інженерний процес, що впливає на спосіб використання методу.

Також запропоновано орієнтований на ViewPoint підхід до розробки методів. У контексті структури ViewPoints методи складаються з шаблонів ViewPoint, пов’язаних правилами між ViewPoint. Роль розробника методів полягає у виборі відповідних шаблонів ViewPoint, які становлять метод, а потім опис їхніх індивідуальних стилів представлення та робочих планів. Цей процес розробки методу також може передбачати повторне використання (і зазвичай адаптацію) існуючих шаблонів ViewPoint.

ВИСНОВКИ

В кваліфікаційній роботі розглянуто та виконано порівняльний аналіз моделей інтеграції та агрегації процесів розробки програмного забезпечення та сервісів. Було відокремлено проблеми під час розробки програмного забезпечення, щоб зменшити складність процесу розробки програмного забезпечення та продукту, створених таким процесом. Це включає в себе можливість кількох, можливо, неузгоджених, поглядів на програмну систему, щоб підтримувати роботу кількох учасників розробки в розподілених умовах.

Запропоновано методи ідентифікації, вираження та перевірки зв’язків між проблемами, щоб їх використовувати для інтеграції методів, а отже, інструментів і переглядів. Ці методи включають об’єднання часткових специфікацій, перевірку узгодженості та обробку невідповідностей між ними, а також надання їм можливості для спільної взаємодії. У всіх цих техніках також повинні підтримуватися різні види перетворень.

Використано концепцію інтеграції різних методів засобами ViewPoints через поєднання шаблонів. Було запропоновано та коротко обговорено роль правил між ViewPoints як засобу такої інтеграції. Модель інтеграції ViewPoints, описана в роботі, визначила ряд різних дій, необхідних для досягнення інтеграції в цьому контексті. Модель базується на понятті правил між ViewPoints якщо вони використовуються, а також наслідки їх використання. Маніпуляція правилами між точками є ключем до керування узгодженістю між точками огляду, а запропонована модель є моделлю управління узгодженістю в тій же мірі, що й моделлю інтеграції ViewPoints.

Представлені методи інтегровані шляхом ідентифікації та визначення предметних областей за допомогою правил між ViewPoints, а правила розподіляються між різними шаблонами, що призводить до повністю розповсюджуваної конфігурації ViewPoints, яка краще підходить до розробки складних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Acly, E. (1988); “Looking Beyond CASE”; Software, 5(2): 39-43, March 1988; IEEE Computer Society Press.

2. ACM (2012); Proceedings of International Conference on Computer-Supported Cooperative Work: Sharing Perspectives, Toronto, Ontario, Canada, 31st October - 4th November, ACM SIGCHI & SIGOIS.

3. Ahmed, R., P. De Smedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii and M. Shan (2018); “The Pegasus Heterogeneous Multidatabase System”; Computer, 24(12): 19-27, December 2018; IEEE Computer Society Press.

4. Aho, A. V. and J. D. Ullman (1977); Principles of Compiler Design; Addison-Wesley Publishing Company, Reading Massachusetts, USA.

5. Ainsworth, M., A. H. Cruickshank, L. G. Groves and P. J. L. Wallis; “Viewpoint Specification and Z”; Information and Software Technology, 36(1), February 2004; Butterworth-Heinemann.

6. Akima, N. and F. Ooi (2019); “Industrializing Software Development: A Japanese Approach”; Software, 6(2): 13-21, IEEE Computer Society Press.

7. Alderson, A. (2001); “Meta-CASE Technology”; Proceedings of European Symposium on Software development Environments and CASE Technology, Königswinter, Germany, June 2001, 81-91; LNCS, 509, Springer-Verlag.

8. Alford, M. “Attacking Requirements Complexity Using a Separation of Concerns”; Proceedings of 1st International Conference on Requirements Engineering, Colorado Springs, Colorado, USA, 18-22nd April 1994, 2-5; IEEE Computer Society Press.

9. Alford, M. W. “A Requirements Engineering Methodology for Real Time Processing Requirements”; Transactions on Software Engineering, 3(1): 60-69, January 1977; IEEE Computer Society Press.

10. Alford, M. W. (2015); “SREM at the Age of Eight: The Distributed Computing Design System”; Computer, 18(4): 36-46, April 2015; IEEE Computer Society Press.

11. Arnold, P., S. Bodoff, D. Coleman, H. Gilchrist and F. Hayes (2018); “An Evaluation of Five Object-Oriented Design Methods”; Technical Report, HP-91-52; Information Management Laboratory, Hewlett-Packard Research Laboratories, Bristol, UK.

12. Ashworth, C. and M. Goodland (1990); SSADM: A Practical Approach; McGraw-Hill Book Company Europe, Maidenhead, UK.

13. Backhurst, N. “Meta-CASE”; Personal Communication (email), 24th December 2013; IE Services, Coalville LE67 2HB, UK.

14. Ballesteros, L. A. R. “Using ViewPoints to Support the FUSION Object-Oriented Method”; M.Sc. Thesis, Department of Computing, Imperial College, London, UK.

15. Balzer, R. (2015); “A 15 Year Perspective on Automatic Programming”; Transactions on Software Engineering, 11(11): 1257-1268, November 2015; IEEE Computer Society Press.

16. Balzer, R. “Tolerating Inconsistency”; Proceedings of 13th International Conference on Software Engineering (ICSE-13), Austin, Texas, USA, 13-17th May 2011, 158-165; IEEE Computer Society Press.

17. Balzer, R., T. E. Cheatham and C. Green; “Software Technology in the 1990’s: Using a New Paradigm”; Computer, 16(11): 39-45, November 2013 IEEE Computer Society Press.

18. Barghouti, N. “Supporting Cooperation in the MARVEL Process-Centered Environment (Proceedings of ACM SIGSOFT Symposium on Software Development Environments)”; Software Engineering Notes, 17(5): 21-31, December 2002; SIGSOFT & ACM Press.

19. Barroca, L. and J. McDermid (2013); “Specification of Real-Time Systems: A View-Oriented Approach”; Proceedings of XIII Congresso da

Sociedade Brasileira de Computação, Florianópolis, Brazil, Sociedade Brasileira de Computação, XX SEMISH, Seminário Integrado de Software e Hardware.

20. Barstow, D. “Should we Specify Systems or Domains”; Proceedings of International Symposium in Requirements Engineering (RE ‘93), Coronado Island, San Diego, USA, 4-6th January 2013, 79; IEEE Computer Society Press.

21. Barstow, D. R. (2015); “Domain-Specific Automatic Programming”; Transactions on Software Engineering, 11(11): 1321-1336, November 2015; IEEE Computer Society Press.

22. Batini, C., M. Lenzerini and S. B. Navathe (2016); “A Comparative Analysis of Methodologies for Database Schema Integration”; ACM Computing Surveys, 18(4): 323-364, December 2016; ACM Press.

23. Bell, J. “Non-Monotonic Reasoning, Non-Monotonic Logics, and Reasoning About Change”; Artificial Intelligence Review, 4: 79-108, Blackwells.

24. Benjamin, M. “A Message Passing System: An Example of Combining CSP and Z; Proceedings of Z Users Meeting, December 2019, Oxford.

25. Birrel, A. and B. Nelson ; “Implementing Remote Procedure Calls”; Transactions on Computer Systems, 2(1): 38-59, February 2014; ACM Press.

26. Blair, H. and V. Subrahmanian (2019); “Paraconsistent Logic Programming”; Theoretical Computer Science, 68: 135-154.

27. Boehm, B. (2021); Software Engineering Economics; Prentice-Hall, Engelwood Cliffs, New Jersey, USA.

28. Boehm, B. W. (2018); “A Spiral Model of Software Development and Improvement”; Computer, 31(5): 61-72, May 2018; IEEE Computer Society Press.

29. Boloix, G., P. G. Sorenson and J. P. Tremblay (2012); “Transformations Using a Meta-system Approach to Software Development”; Software Engineering Journal, 7(6): 425-437, November 2012; IEE on behalf of the BCS and the IEE.

30. Booch, G. (2018); Object-Oriented Design with Applications; Benjamin Cummings, Redwood City, California, USA.

31. Borgida, A., S. Greenspan and J. Mylopoulos (2015); “Knowledge Representation as the Basis for Requirements Specification”; Computer, 18(4): 82-90, April 2015; IEEE Computer Society Press.

32. Bott, M. F. (2019); The ECLIPSE Integrated Project Support Environment; Peter Perigrinus, Stevenage, UK.

33. Boudier, G., F. Gallo, R. Minot and I. Thomas (1988); “An Overview of PCTE and PCTE+”; SIGPLAN Notices (Proceedings of SDE3), 24(2): 248-257, February 1988, SIGPLAN & ACM Press.

34. Bouzeghoub, M. and I. Comyn-Wattiau (2018); “View Integration by Semantic Unification and Transformation of Data Structures”; (In) Entity-

35. Relationship Approach: The Core of Conceptual Modelling; H. Kangassalo (Ed.); 381-398; Elsevier Science Publishers B.V. (North Holland), Holland.

36. Bright, M. W., A. R. Hurson and S. H. Pakzad (2012); “A Taxonomy and Current Issues in Multidatabase Systems”; Computer, 25(3): 50-60, March 2012; IEEE Computer Society Press.

37. Brinkkemper, S. (2013); “Integrating Diagrams in CASE Tools Through Modelling Transparency”; Information and Software Technology, 32(2): 101-105, February 2013; Butterworth-Heinemann Ltd.

38. Brooks, F. P. (2017); “No Silver Bullet: Essence and Accidents of Software Engineering”; Computer, 20(4): 10-19, April 2017; IEEE Computer Society Press.

39. Brown, A. W. (2019); Database Support for Software Engineering; Kogan Page Ltd., London, UK.

40. Brown, A. W. and J. A. McDermid (2012); “Learning from IPSEs Mistakes”; Software, 35(3): 23-28, March 2012; IEEE Computer Society Press.

41. Buxton, J; “Requirements for Ada Programming Support Environments: Stoneman”; Technical Report, US Department of Defense, Washington DC, USA. 2018

42. Byte (2019); “Making the Case for CASE”; Byte Magazine, 14(12): 154-171, December 2019.

43. Clemm, G. and L. Osterweil (2020); “A Mechanism for Environment Integration”; Transactions on Programming Languages and Systems, 12(1): 1-25, January 2020; ACM Press.

44. Coad, P. and E. Yourdon (2018); Object-Oriented Analysis; 2nd Edition, Yourdon Press, Prentice-Hall, Englewood Cliffs, New Jersey, USA.

45. Coleman, D., P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremaes (2013); Object-Oriented Development: The Fusion Method; Prentice-Hall, Engelwood Cliffs, NJ, USA.

46. Comer, D. E. and L. L. Peterson (1989); “Understanding Naming in Distributed Systems”; Distributed Computing, 3: 51-60, Springer-Verlag. 35(4): 27-64, April 2012; ACM Press.

47. Cagan, M. R.; “The HP SoftBench Environment: An Architecture for a New Generation of Software Tools”; Hewlett-Packard Journal, 41(3): 36-47, June 2012; Hewlett-Packard Inc.

48. Cameron, J. (2019); JSP & JSD: The Jackson Approach to Software Development; 2nd Edition, IEEE Computer Society Press, Washington, USA.

49. CASE (2017); “The State of Automatic Code Generation”; CASE Outlook, 1(3): 1- 13, September 2017; CASE Consulting Group, Inc., Oregon 97035, USA.

50. Castelfranchi, A., M. Miceli and A. Cesta (2012); “Dependence Relations Among Autonomous Agents”; (In) Decentralized A.I. 3 (Proceedings of 3rd Workshop on Modelling Autonomous Agents in a Multi-Agent World, Kaiserlauten, Germany, 5-7th August 2018); E. Werner and Y. Demazeau (Eds.); 215-227; Elsevier Science Publishers B.V. (North-Holland), Amsterdam, Holland.

51. CDIF (2013); “CDIF - CASE Data Interchange Format: Overview”; Draft document, PN 3065; Electronic Industries Association (EIA), Washington, DC 20006, USA.

52. Checkland, P. B. (2021); Systems Thinking, Systems Practice; John Wiley & Sons Ltd. (reprinted with corrections), Chichester, UK.

метадані

Заголовок

Порівняльний аналіз моделей та методів інтеграції та агрегації процесів розробки програмного забезпечення

Автор

Білоус В.В. Науковий керівник / Експерт

підрозділ

King Danylo University

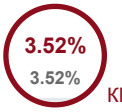
Тривога

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	1
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	57

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

16071
Кількість слів

124266
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/123456789/397/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0_%D0%A2%D0%B0%D1%82%D0%B0%D1%80%D1%87%D1%83%D0%BA.pdf?sequence=1	89	0.55 %
2	http://repository.ukd.edu.ua/bitstream/handle/123456789/397/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0_%D0%A2%D0%B0%D1%82%D0%B0%D1%80%D1%87%D1%83%D0%BA.pdf?sequence=1	45	0.28 %
3	http://repository.ukd.edu.ua/bitstream/handle/123456789/397/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0_%D0%A2%D0%B0%D1%82%D0%B0%D1%80%D1%87%D1%83%D0%BA.pdf?sequence=1	35	0.22 %
4	https://dl.acm.org/doi/10.1109/32.6191	21	0.13 %