КВАЛІФІКАЦІЙНА РОБОТА

Група МІПЗс-22

Бельмега Р.Я.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Бельмега Ростислав Ярославович

УДК 004.4

Підвищення семантичної ефективності моделей групової динаміки команд розробників програмного забезпечення

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль	Студент
_____________Стисло О.В.	_____________ Бельмега Р.Я.
(підпис, дата, розшифрування підпису)	(підпис, дата, розшифрування підпису)
Допускається до захисту Завідувач кафедри	Керівник роботи
_____________к.т.н., доц. Ващишак С.П.	_____________ к.ф-м.н, доц. Бойчук А.М
(підпис, дата, розшифрування підпису)	(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «магістр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

____

« 19 » лютого 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Бельмезі Ростиславу Ярославовичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Підвищення семантичної ефективності моделей групової динаміки команд розробників програмного забезпечення

керівник роботи:

Бойчук Андрій Михайлович, кандидат фізико-математичних наук, доцент

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

2. Термін подання студентом роботи 16.02.2024

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

4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)
   1. Аналіз структури процесу командної розробки програмного забезпечення
   2. Моделі та шаблони групової динаміки команд розробників програмного забезпечення
   3. Підвищення семантичної ефективності моделей шляхом розробки фреймворку середовища командної розробки

5. Дата видачі завдання 29.06.2023

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

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

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Аналіз структури процесу командної розробки програмного забезпечення	26.09.2023	Виконано
2.	Дослідження моделей та шаблонів групової динаміки процесу розробки	20.10.2023	Виконано
3.	Реалізація семантики фреймворку середовища розробки	15.11.2023	Виконано
4.	Розробка моделі та модифікації артефакту при використанні фрейворку командної розробки	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 ____________________ Бельмега Р.Я.

(підпис) (прізвище та ініціали)

Керівник роботи ____________________ Бойчук А.М.

(підпис) (прізвище та ініціали)

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

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу
14	Типова конфігурація процесу розробки програмного забезпечення	39	Карта шаблонів CSE
19	Сценарій розробки програмного забезпечення з використанням інструментів CSE на основі Caise	51	Діаграма класів UML для спрощеної семантичної моделі програмного забезпечення
21	Класичний вигляд життєвого циклу розробки програмного забезпечення	53	Комбіноване середовище для двох розробників, використовуючи нотацію UML

22	Типове дерево історії версій для проекту розробки програмного забезпечення	56	Загальне схематичне зображення структури Caise
23	Модифікації артефактів у проекті розробки програмного забезпечення	58	Модифікація артефакту в рамках Caise
26	Інструмент UML для спільної роботи	62	Структура Caise в контексті спільного використання
27	Веб-інструмент для спільного проектування документів Rosetta	63	Представлення фреймворку Caise та основних складових інструментів
28	Середовище спільної розробки XP Moomba	64	Зв'язки між ключовими компонентами структури Caise
29	Редактор коду для спільної роботи Tukan	65	Семантична модель ОО забезпечення в нотації UML
29	Інструмент редагування графіка в Eclipse Communication Framework	69	Три концептуальні рівні структури Caise
30	Інструмент перевірки Augur	70	Схематичний вигляд модифікації артефакту в Caise з сервером Caise
31	Інструмент спільної візуалізації Palantír	72	Модель події Caise
35	Приклад шаблону «Дія/реакція»	73	Ключові типи дій у рамках Caise протоколу

АНОТАЦІЯ

Кваліфікаційна робота присвячена дослідженню процесів підвищення семантичної ефективності моделей групової динаміки команд розробників програмного забезпечення шляхом розробки фреймворку на основі семантичної моделі.

В першому розділі виконано аналіз структури процесу командної розробки програмного забезпечення, досліджено концепції групової динаміки при розробці програмного забезпечення та сутність процесу командної розробки.

В другому розділі представлено моделі та шаблони групової динаміки команд розробників програмного забезпечення, досліджено інструменти для процесу спільної розробки програмного забезпечення. Наведено приклад реалізації концепції шаблонів групової динаміки процесів розробки і виконано побудову карт шаблонів для групової розробки.

В третьому розділі виконано підвищення семантичної ефективності моделей шляхом розробки фреймворку середовища командної розробки. Запропоновано архітектуру фреймворку для спільної командної розробки програмного забезпечення, виконано архітектурне проектування та розробку семантичної моделі фреймоворку командної розробки. Проведено подання моделі події та модифікації артефакту при використанні фрейворку командної розробки.

КЛЮЧОВІ СЛОВА: МОДЕЛЬ ШАБЛОНІВ, СПІЛЬНА РОЗРОБКА, ГРУПОВА ДИНАМІКА, РЕФАКТОРИНГ, ФРЕЙМВОРК РОЗРОБКИ, ІТЕРАЦІЯ, ЕФЕКТИВНІСТЬ ПРОЦЕСУ, СЕМАНТИЧНА МОДЕЛЬ.

SUMMARY

The qualification work is devoted to the research of the processes of increasing the semantic efficiency of models of group dynamics of teams of software developers by developing a framework based on a semantic model.

In the first chapter, an analysis of the structure of the team software development process was performed, the concepts of group dynamics during software development and the essence of the team development process were investigated.

The second chapter presents models and patterns of group dynamics of software development teams, explores tools for the process of joint software development. An example of the implementation of the concept of templates of group dynamics of development processes is provided, and the construction of template maps for group development is performed.

In the third section, the semantic efficiency of the models was increased by developing the framework of the team development environment. The architecture of the framework for joint team development of software was proposed, the architectural design and development of the semantic model of the team development framework was performed. The presentation of the event model and the modification of the artifact when using the framework of team development was carried out.

KEY WORDS: TEMPLATE MODEL, COLLABORATIVE DEVELOPMENT, GROUP DYNAMICS, REFACTORING, DEVELOPMENT FRAMEWORK, ITERATION, PROCESS EFFICIENCY, SEMANTIC MODEL.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 9
ВСТУП 10
РОЗДІЛ 1. АНАЛІЗ СТРУКТУРИ ПРОЦЕСУ КОМАНДНОЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 13
1.1 Опис предметної області дослідження 13
1.2 Дослідження концепції групової динаміки при розробці програмного забезпечення 17
1.3 Дослідження процесу командної розробки програмного забезпечення 20
Висновки до розділу 1 24
РОЗДІЛ 2. МОДЕЛІ ТА ШАБЛОНИ ГРУПОВОЇ ДИНАМІКИ КОМАНД РОЗРОБНИКІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 25
2.1 Дослідження інструментів для процесу спільної розробки програмного забезпечення 25
2.2 Приклад реалізації концепції шаблонів групової динаміки процесів розробки 32
2.3 Побудова карт шаблонів для групової розробки 38
Висновки до розділу 2 48
РОЗДІЛ 3. ПІДВИЩЕННЯ СЕМАНТИЧНОЇ ЕФЕКТИВНОСТІ МОДЕЛЕЙ ШЛЯХОМ РОЗРОБКИ ФРЕЙМВОРКУ СЕРЕДОВИЩА КОМАНДНОЇ РОЗРОБКИ 50
3.1 Розробка програмного забезпечення на основі семантичної моделі 50
3.2 Архітектура фреймворку для спільної командної розробки програмного забезпечення 55
3.3 Архітектурне проектування та розробка семантичної моделі фреймоворку командної розробки 62

3.4 Подання моделі події та модифікації артефакту при використанні фрейворку командної розробки 71
Висновки до розділу 3 77
ВИСНОВКИ 78
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 79

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CSE - Collaborative Software Engineering

CSCW - Computer Supported Collaborative Work

SE - Software Engineering

CAISE - Collaborative Architecture for Iterative Software Engineering

CRC - Class-ResponsibilityCollaborators

ПЗ – програмне забезпечення

ООПЗ – об’єктно-орієнтоване програмне забезпечення

ВСТУП

Актуальність теми дослідження.и Останнім часом компонентне програмне забезпечення розглядається як життєздатна та економічна альтернатива традиційному процесу розробки програмного забезпечення. Здатність будувати повні системні рішення шляхом з'єднання через незалежно створені загальнодоступні інтерфейси розгорнутих компонентів, є рушійною силою успіху компонентної програмної інженерії (CBSE).

Основна неоднорідність розподілених обчислень, разом із проблемами, які виникли під час реалізації подібних систем в порівнянні зі звичайним розробкою програмного забезпечення викликало потребу в програмному забезпеченні на основі компонентів. CBSE вирішує не лише складності розподілених обчислень, воно також відіграє все більшу роль у загальній розробці програмних систем. Очікується, що компоненти зменшать вартість і час виходу на ринок систем, частиною яких вони є, і підвищать їх якість. В останні роки спостерігається значне зростання програмного забезпечення інженерії аспектів виробництва та використання комплектуючих. Повторне використання компонентів програмного забезпечення може значно знизити як витрати на розробку програмного забезпечення, так і також життєвий цикл розробки. У результаті час виходу на ринок знижується в основному через те, що складності, пов'язані з великими гетерогенними системами вирішуються використанням готових компонентів, спеціалізованих для цього завдання.

Компоненти забезпечують гнучкість способу транспортування функцій у зручній упаковці чорної скриньки, яка спрощує розповсюдження рішення. Компоненти, хоча вони вимагають упаковки, щоб використовувати їх, вони тим не менше більш гнучкі та простіші для повторного використання, ніж інші подібні підходи повторного використання , наприклад, шаблони дизайну. Цей аргумент знаходить багато уваги останнім часом, особливо у великих системах. Якщо компонент використовували кілька разів, ймовірно є

надійнішим, ніж інше програмне забезпечення, розроблене з «нуля», оскільки воно було випробувано в більшій різноманітності умов.

Проте, основною проблемою CBSE є якість компонентів, що використовуються в системі. Загальновідомо, що сила ланцюга дорівнює силі найслабшого посилання. Подібно до цієї концепції, надійність програмної системи на основі компонентів залежить від надійності компонент, з яких вона виготовлена. Тому в CBSE, належно розглядається процес пошуку та відбору компонентів, як наріжний камінь для розробки будь-якої ефективної системи на основі компонентів. Досі індустрія програмного забезпечення була зосереджена на функціональних аспектах компонентів, відходячи осторонь складного завдання оцінки їх якості. Якщо забезпечення якості програмного забезпечення, розробленого власними силами, є складним завданням, виконання програмного забезпечення, розробленого в іншому місці, часто без доступу до вихідного коду та детальної документації викликає ще більші складності відповідно. Можливе впровадження програмних компонентів невідомої якості може мати катастрофічні результати.

Мета і завдання дослідження. Метою роботи є імплементація та підвищення ефективності семантичних моделей процесів розробки програмного забезпечення в контексті групової динаміки даного процесу.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

– виконати аналіз структури процесу розробки програмного забезпечення;

– здійснити дослідження концепції групової динаміки при розробці програмного забезпечення;

– виконати побудову моделей та шаблонів групової динаміки процесу розробки;

– побудувати карти шаблонів для групової розробки програмного забезпечення;

– реалізувати семантики фреймворку середовища розробки;

– виконати архітектурне проектування та розробку семантичної моделі фреймоворку командної розробки.

Об'єктом дослідження є архітектура та методи побудови фреймворку для спільної командної розробки програмного забезпечення.

Предмето дослідження є підвищення семантичної ефективності моделей групової динаміки команд розробників програмного забезпечення.

Методи дослідження базуються на використанні методів програмної інженерії для задач моделювання групової динаміки команд розробників та семантичних аспектів оцінки програмних компонент.

Наукова новизна одержаних результатів полягає у тому, що на основі ґрунтовного аналізу досліджуваної області було розроблено концепцію групової динаміки при розробці програмного забезпечення шляхом реалізації концепції шаблонів групової динаміки процесів розробки та побудови карт шаблонів для групової розробки та архітектурного проектування семантичної моделі розробки.

Практичне значення одержаних результатів полягає в запропонованій структурі фреймоврку спільної розробки CAISE, що забезпечує хороший рівень співпраці в рамках розробки програмного забезпечення шляхом семантичного моделювання, адаптації нових мов та інструментів, підтримки масштабованості та можливості налаштування та розширення.

Структура. Кількість розділів – 3. Загальний обсяг основної частини – 84 сторінок. Список використаних джерел містить – 50 позицій.

РОЗДІЛ 1. АНАЛІЗ СТРУКТУРИ ПРОЦЕСУ КОМАНДНОЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Опис предметної області дослідження

Розробка програмного забезпечення ПЗ (SE - Software Engineering), є командним процесом. Розробники проекту ПЗ працюють разом на всіх етапах життєвого циклу розробки програмного забезпечення. У будь-якій сфері працювати в команді важко через необхідність постійного спілкування та координації завдань. Співпраця в групах розробників програмного забезпечення ще більше ускладнюється завданням підтримки ряду продуктів, кожен з яких має кілька версій.

Інструменти ПЗ зазвичай погано підтримують співпрацю. Замість того, щоб бути розробленими на основі спільних процесів, центральних для ПЗ, інструменти базуються на погляді одного користувача на життєвий цикл розробки. Поширені проблеми, які не вирішують поточні інструменти, включають транзакційні конфлікти, коли одночасні зміни в проекті конфліктують один з одним семантично, особливо конфлікти при інтеграції, коли одночасні зміни в тому самому вихідному файлі конфліктують лексично. Обидві ці проблеми виникають через погану обізнаність про дії інших користувачів і нездатність синхронізувати роботу розробників на всіх рівнях.

Ці проблеми посилюються типовою ідіомою копіювання, модифікації та злиття звичайних систем керування вихідним кодом, де довгі інтервали ізольованої розробки є звичайними, що збільшує труднощі процесу інтеграції вихідного коду та створення. Типова конфігурація для звичайної розробки програмного забезпечення представлена на рисунку 1.1, де сповіщення про модифікації програми отримуються переважно через системи сховищ вихідного коду, а підтримка інструментів для спілкування між розробниками є низькою.

Стверджується, що чим кращий зв’язок і краща співпраця, тим кращим буде процес розробки ПЗ з точки зору продуктивності та якості кінцевого продукту.

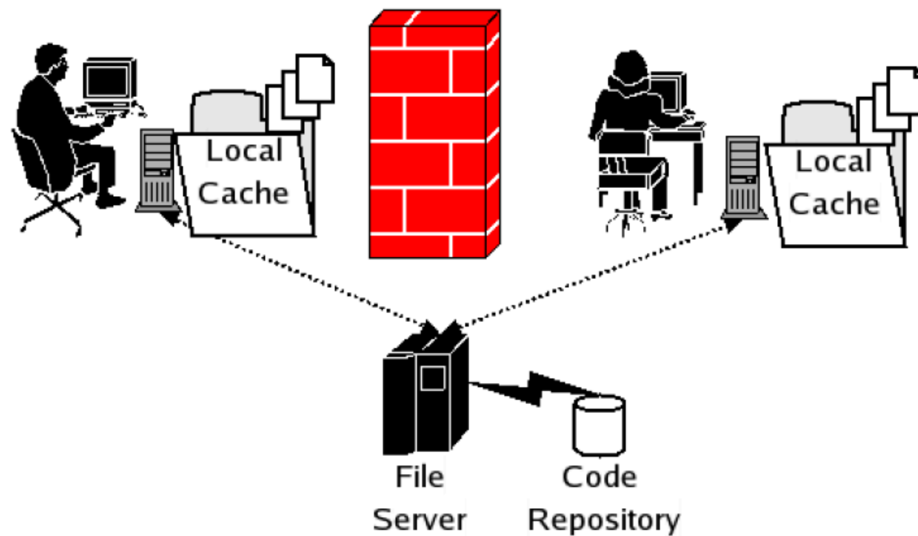


Рисунок 1.1 – Типова конфігурація процесу розробки програмного забезпечення

Однак, оскільки сфера спільної розробки програмного забезпечення (CSE - Collaborative Software Engineering;) знаходиться в початковому стані, багато з цих тверджень ще потрібно перевірити емпірично. Лише після того, як з’являться якісні інструменти реального часу, дослідники зможуть реалізувати весь потенціал і переваги CSE.

Підтримка співпраці, як синхронної, так і асинхронної, існує в інших областях, не пов’язаних безпосередньо з ПЗ. В офісному середовищі можна певною мірою ділитися документами та спільно редагувати їх з колегами. Засоби для телеконференцій також є звичним явищем сьогодні, як і віртуальні зустрічі в реальному часі, де спільні віртуальні дошки використовуються для передачі ідей у групі розподіленого персоналу.

Успішне застосування асинхронної та синхронної співпраці в інших галузях досліджень свідчить про те, що практика SE також може стати більш орієнтованою на співпрацю. На жаль, інструменти для підтримки CSE важко

розробити; SE включає аспекти, які складно врахувати за допомогою поточної технології спільної роботи з комп'ютерною підтримкою (CSCW - Computer Supported Collaborative Work). Зокрема, ПЗ базується на численних складних артефактах, таких як вихідні файли, де синтаксис документа розширений і жорсткий, а також існує багато зв'язків між артефактами. Хоча додавання функцій CSCW до існуючих однокористувацьких інструментів є інтуїтивно зрозумілим першим кроком до нових інструментів CSE, це не обов'язково масштабує чи забезпечує передбачуваний рівень покращення.

Як приклад типових викликів CSCW у CSE, що слід робити, коли користувач наполовину використав тіло методу в текстовому редакторі, а користувач у діаграмі UML перейменовує цей конкретний метод ? Чи всі зміни коду втрачено? Якщо ні, як можна зберегти зміну коду ? Тому більшість функцій спільної роботи як для комерційних інструментів, так і для дослідницьких прототипів обмежують співпрацю до парного програмування та механізмів контролю передачі токенів.

Будь-який інструмент CSE має складну структуру, як наприклад розробка інтерфейсу користувача, контроль і управління шарами CSCW, різні рівні вимог до співпраці, різні очікування між розробниками всередині групи, підтримка кількох типів артефактів і потенційно кілька переглядів артефактів. Існують також технічні аспекти, які необхідно розглянути, такі як керування паралелізмом і дизайн розподіленої системи, а також стандартні технічні аспекти ПЗ, такі як аналіз, семантичне моделювання та керування вихідним кодом. Відповідно, лише кілька дослідницьких прототипів, таких як Poseidon для UML [2] перетворилися на професійні інструменти.

Незважаючи на те, що, можливо впровадити інструменти ПЗ, розширені для співпраці, єдиною суттєвою перешкодою для успіху впровадження таких інструментів може бути погане співвідношення потужності інструментів і зусиль у розробці. Навіть після того, як буде розроблено якісний інструмент CSE, немає гарантії, що він отримає широке впровадження через різні вимоги розробників програмного забезпечення.

Метою дослідження в цій роботі є дослідження механізмів підтримки CSE у реальному часі та визначення переваг для розробників програмного забезпечення при використанні таких інструментів.

Дослідження в цій роботі зосереджено на створенні основи та прототипів фреймворків для інструментів CSE. Структура, Caise - Collaborative Architecture for Iterative Software Engineering (спільна архітектура для ітеративної розробки програмного забезпечення), підтримує розробку інструментів CSE, які працюють як синхронно, так і асинхронно. Ці типи інструментів CSE можуть бути розроблені таким чином, щоб уникнути проблем, пов'язаних зі звичайними підходами до розробки програмного забезпечення, шляхом підвищення комунікації програмістів і поінформованості про дії інших.

Фреймворк Caise дозволяє програмістам працювати разом. Синхронні інструменти CSE на основі Caise досягають цього, підтримуючи всіх програмістів у групі синхронізованими в режимі реального часу, водночас надаючи настроювану обізнаність користувача та інформацію про стан проекту для окремих інструментів. Фреймворк Caise найкраще підходить для невеликої групи розробників, які бажають працювати разом і в тісному контакті над усім програмним проектом. Структура Caise забезпечує інфраструктуру з потенціалом для підтримки всього процесу розробки. Можна створювати інструменти на основі Caise, які забезпечують більше, ніж просто спільне редагування базових програмних артефактів. Спільну компіляцію, тестування та налагодження проектів програмного забезпечення також можна реалізувати за допомогою послуг Caise. Також можуть бути побудовані комплексні комунікаційні засоби між розробниками.

За допомогою цього фреймворку було створено багато інструментів, зокрема редактори для спільної роботи та створення діаграм, багаторазові системи CSE, які можна додати до будь-якого існуючого інструменту розробки, нові типи візуалізації активності користувачів на основі детальної журналізації та інформації про структуру проекту, а також користувацькі CSE

такі інструменти, як агенти для управління проектами розробки програмного забезпечення в реальному часі.

1.2 Дослідження концепції групової динаміки при розробці програмного забезпечення

Щоб проілюструвати концепцію структури Caise, розглянемо досить поширений сценарій розробки ПЗ. По-перше, наведено приклад конфлікту кодування та вирішення між двома розробниками за допомогою звичайних інструментів. Далі розглянемо інший приклад того самого сценарію, але вже з підтримкою інструментів CSE на основі Caise.

Передумови розробки. Розробники Боб і Аліса працюють над одним проектом, розробляючи графічний редактор із простим інтерфейсом користувача, написаним на Java. Код інтерфейсу користувача міститься в одному файлі під назвою GUI.java , а код для збереження файлу знаходиться у файлі під назвою Persistence.java . Вимагають завершення два завдання: заміна виклику методу save(int fileType) на виклик методу saveXML() із файлу GUI.java та додавання нового методу під назвою save() до Persistence.java , який зберігає поточний файл за допомогою стандартний системний формат.

Використання звичайних інструментів. Використовуючи звичайні текстові редактори та сховище коду, наприклад CVS, Боб і Аліса візьмуть окремі копії поточної версії коду зі сховища (наприклад, версії 1.1) і почнуть працювати незалежно. Аліса вирішила відредагувати код GUI і замінила виклик методу save(int fileType) на saveXML(). У цей момент Аліса повторно компілює свій код, тестує свою робочу копію програми, щоб переконатися, що вона працює, а потім повертає свої файли в репозиторій CVS. Це позначає GUI.java як версію 1.2.

У той же час Боб починає працювати над своїм завданням із додавання нового методу save() до файлу Persistence.java. Він не тільки завершує цей

метод, але також видаляє існуючий метод saveXML(), оскільки згідно з поточною кодовою базою (версія 1.1) виклики цього методу не здійснюються. Боб повторно компілює свою програму і вона компілюється та виконується без помилок. Потім він повертає свої файли в репозиторій, який позначає Persistence.java як версію 1.2.

Потім і Боб і Аліса залишають на день, знаючи, що вони успішно вдосконалили останню версію інструменту для побудови графіків. На жаль, коли вони приходять на роботу наступного дня, їм повідомляють, що нічне відновлення зі сховища коду не вдалося через невирішений виклик методу saveXML().

Призначивши зустріч, Боб і Аліса усвідомлюють джерело проблеми, яку вони ненавмисно створили. Щоб вирішити цю проблему, вони вирішують, що для збереження в XML слід викликати новий метод save() після встановлення формату системного файлу на XML. Тому Аліса знову перевіряє всю кодову базу (версія 1.2), змінює виклик у GUI.java з saveXML() на save() після визначення формату системного файлу та повторно компілює програму. Після перевірки роботи програми GUI.java повертається до центрального репозиторію як версія 1.3 і тепер проект оновлений.

Працюючи в співпраці один з одним, Боб і Аліса можуть відразу помітити, що їхні завдання можуть конфліктувати, якщо не вжити певних заходів, і що для успішної зміни існуючої бази коду необхідні певні комунікації. Однак у звичайній практиці кодування конфлікти такого характеру є звичайним явищем, особливо коли кількість одночасних розробників зростає. Протоколи, які використовує команда програмістів, такі як частота інтеграції коду та ступінь зв’язку під час розробки, регулюють кількість конфліктів, що виникають, і рівень зусиль, необхідних для об’єднання суперечливих модифікацій.

Використання інструментів для спільної роботи. Використовуючи інструменти CSE у реальному часі на основі Caise для виконання вищевказаного завдання, взаємодія розробників може бути дуже різною.

Незважаючи на те, що Боб і Аліса можуть розпочати роботу над окремими завданнями, використовуючи різні типи інструментів розробки, як тільки Боб і Аліса розмістять свої курсори в семантично пов'язаному коді, структура Caise надішле їхнім інструментам сповіщення про існування семантичного зв'язку. Інструмент CSE на основі Caise, який реагує на цей тип зворотного зв'язку, подано на рисунку 1.2.

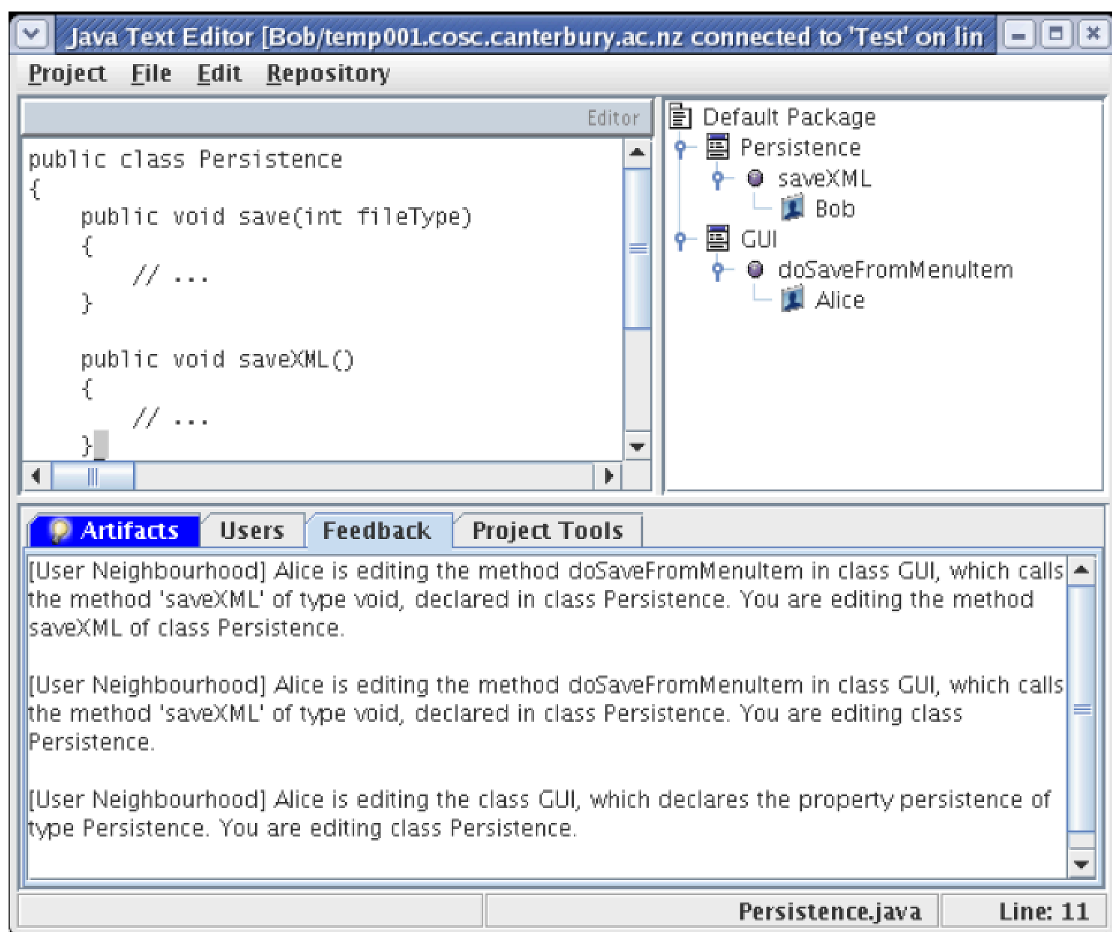


Рисунок 1.2 – Сценарій розробки програмного забезпечення з використанням інструментів CSE на основі Caise

Використовуючи поточний сценарій кодування, коли Боб наводить курсор на метод saveXML() з наміром видалити його, його інструмент зможе повідомити, що цей метод тепер викликається з методу у файлі GUI.java, який є наразі редагується Алісою. Ця інформація відображається на панелі зворотного зв'язку, що представлено в нижній частині рисунка 1.2. Навіть

якщо Боб вирішив видалити цей метод, і він, і Аліса будуть сповіщені, що програму щойно було змінено через невирішений виклик методу.

Це лише простий приклад можливостей інструментів на основі Caise для групової та спільної роботи. В наступному розділі детально представлено деякі типові інструменти на основі Caise, які забезпечують багато різних типів механізмів зворотного зв'язку, наприклад показники реального часу, засоби чату та віджети, які вказують на поточний стан проекту та пов'язані з ним артефакти, такі як вихідні файли. У наведеному вище сценарії наведено чітку ілюстрацію фундаментальної передумови структури Caise: завдяки моніторингу активності користувача в режимі реального часу та семантичному аналізу програмного забезпечення, що лежить в його основі, можна забезпечити набагато більшу підтримку співпраці, ніж доступна зараз у звичайних інструментах розробки ПЗ.

1.3 Дослідження процесу командної розробки програмного забезпечення

Класичну модель для абстрактного представлення процесу розробки програмного забезпечення представлено в [4] подано рисунку 1.3. Ця модель показує процес розробки від отримання рішення на основі абстрактного аналізу та забезпечує кінцевий робочий продукт у формі програмного забезпечення.

Оскільки дана модель є загальною, вона не посилається на конкретні фактори розробки, які слід враховувати під час розробки програмного забезпечення. Основні фактори будь-якого проекту ПЗ включають вихідні файли та версії, використовувані мови програмування, версії продукту та гілки, команди для кожного етапу розробки та методології розробки, яких дотримуються під час роботи.

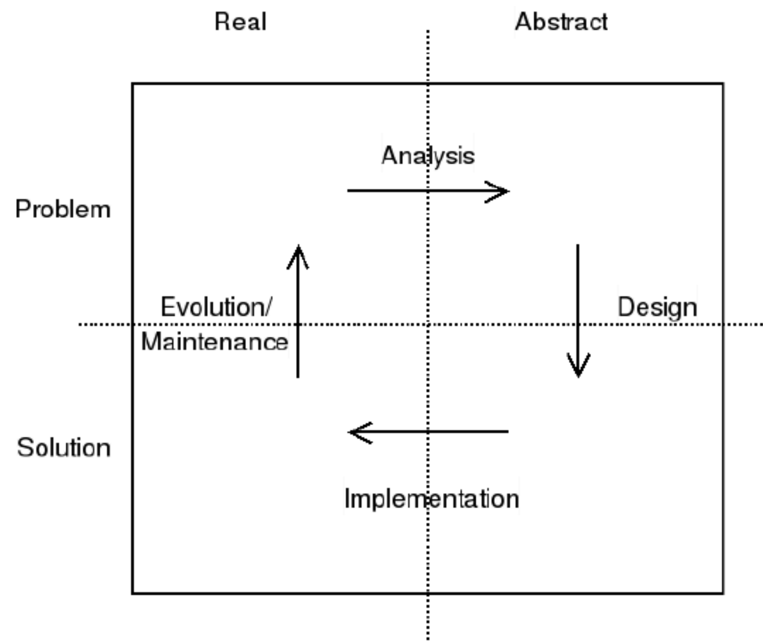


Рисунок 1.3 – Класичний вигляд життєвого циклу розробки програмного забезпечення

Еволюція процесів SE стає все більш важливою. Завдяки великим фреймворкам і бібліотекам, компонентному ПЗ та мовам програмування з методами повторного використання та «програмування за відмінностями», є звичайним явищем для додатків, які підлягають рефакторингу, перепроектуванню, повторному використанню та навіть об'єднанню з іншими продуктами. Таким чином, рисунок 1.3 було вдосконалено додатковою фазою еволюції, яка вказує на те, що реалізації можуть просто представляти кінець однієї ітерації циклу.

Деякі типи артефактів ПЗ характерні для певних фаз поданої моделі. Наприклад, документація щодо вимог зазвичай формується виключно на етапі аналізу, оскільки вона, переважно, буде використана лише як довідковий матеріал для решти етапів. Для інших артефактів, таких як діаграми класів, вони можуть використовуватися для кількох або всіх фаз моделі протягом будь-якої заданої ітерації.

Для будь-якого проекту розробки програмного забезпечення версії вихідних файлів зберігаються в системі сховища вихідного коду. Це

незалежно від кількості програмістів або методології розробки. Завдяки розгалуженню, як показано на рисунку 1.4, незначні модифікації файлів у попередній версії програми можна внести одразу за запитом, незалежно від стану компіляції поточної версії проекту. Такі зміни потім можна буде інтегрувати в основну версію продукту, коли вона буде в стані компіляції та стабільності, замість того, щоб працювати над створенням поточної версії проекту, яка також включає нещодавно запитані модифікації.

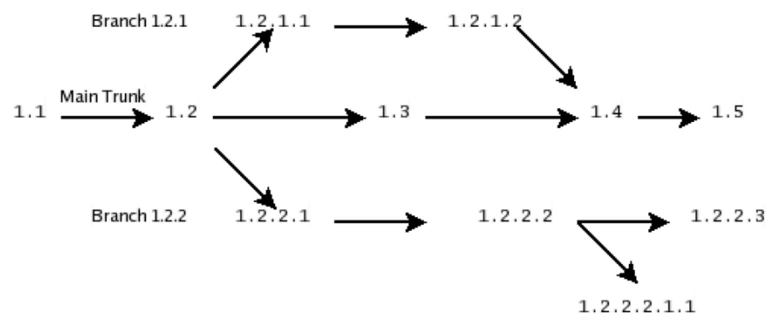


Рисунок 1.4 – Типове дерево історії версій для проекту розробки програмного забезпечення

Розгалуження, коли повний набір файлів проекту дублюється для альтернативного потоку розробки, не обов’язково має здійснюватися в рамках проекту розробки програмного забезпечення. Однак у межах одного каналу розробки системи репозиторіїв вихідного коду все одно будуть перевіряти послідовні версії всіх файлів проекту, переважно робиться автоматично після кожного повернення змінених файлів у центральне сховище. Відповідно, важливо розуміти, що системи сховищ вихідного коду мають можливість створювати попередню версію будь-якого файлу, можливо, замінюючи поточну версію, якщо потрібно.

Враховуючи те, що може існувати кілька гілок проекту, і що кожен файл у гілки має потенційно велику кількість попередніх версій, то увага має бути зосереджена на життєвому циклі файлу в контексті однієї версії.

Будь-яка версія одного файлу може бути змінена кількома розробниками одночасно. Під час використання звичайних інструментів файли зазвичай поділяються за допомогою ідіом копіювання, модифікації та злиття, відповідно кожен користувач отримує копію поточної версії файлу, вносить свої зміни і після внесення всіх змін набір змінених файлів об’єднується в одну нову версію.

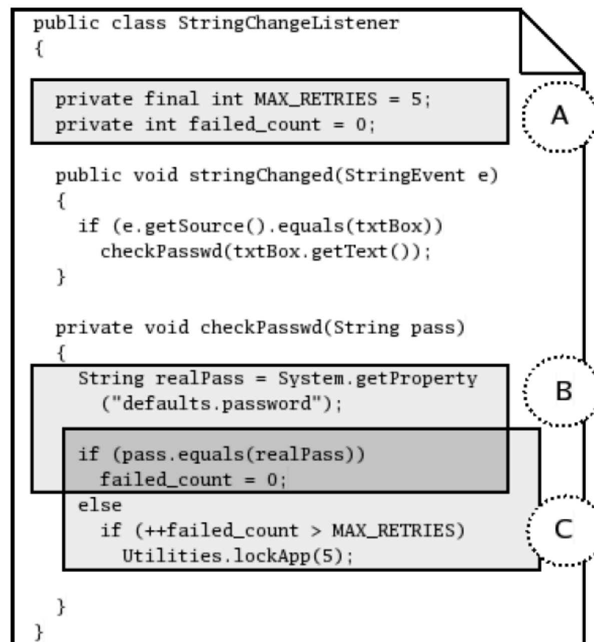


Рисунок 1.5 – Модифікації артефактів у проекті розробки програмного забезпечення

Одна версія файлу може зазнавати складних одночасних змін. Використовуючи рисунок 1.5 для ілюстрації, зміна в області A не створює ймовірних проблем модифікації. Однак для B і C дуже ймовірно, що будь-які одночасні зміни конфліктуватимуть під час об’єднання. Навіть якщо синтаксис цих двох змін безпосередньо не заважає одна одній, ймовірно, що лексична близькість цих змін все одно спричинить конфлікт злиття, коли нова версія файлу не може бути створена автоматично.

Процес об’єднання файлів зазвичай виконується посимвольно на основі символів, де не потрібно виконувати аналіз синтаксису чи семантики

змінених вихідних файлів. Після того, як одночасні модифікації вихідного файлу об'єднано в нову версію, отриманий файл закріплюється в основному репозиторії та розповсюджується серед усіх розробників.

Однак злиття не є тривіальним процесом. Навіть найновіші засоби автоматичного об'єднання файлів не можуть вирішити всі, крім найпростіших завдань об'єднання файлів. Це залишає розробникам завдання виправлення змін, які конфліктують, що може бути трудомістким процесом.

Очевидно, що життєвий цикл будь-якого артефакту в рамках проекту є складним. Незалежно від конфліктів злиття всередині файлу, ще одна проблема існує під час одночасної модифікації набору файлів. Ця проблема, яку можна назвати транзакційним конфліктом який виникає, коли змінений файл, незважаючи на синтаксичну правильність призводить до неминучої помилки збірки проекту через пошкоджену залежність коду.

У звичайних інструментах користувачі не виявлять конфлікти транзакцій, доки всі нові версії змінених файлів не будуть об'єднані з центральним сховищем і не буде зроблена спроба перебудувати проект.

Конфлікти транзакцій можна розглядати на рівні проекту, а не на рівні файлу. Однак, обговорюючи життєві цикли артефактів, важливо розуміти, що конфлікти транзакцій впливають на артефакти — залучені артефакти потребують подальшої модифікації, щоб уможливити створення проекту знову, навіть якщо це означає повернення до попередньої версії файлу.

Висновки до розділу 1

В даному розділі проведено огляд процесу командної розробки програмного забезпечення, основні концепції групової динаміки при розробці та технології модифікації та злиття версійностей проектів розробки.

РОЗДІЛ 2. МОДЕЛІ ТА ШАБЛОНИ ГРУПОВОЇ ДИНАМІКИ КОМАНД РОЗРОБНИКІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Дослідження інструментів для процесу спільної розробки програмного забезпечення

Рушійними факторами сучасного розвитку розробки програмного забезпечення є поява потужних IDE з відкритим кодом, зміцнення стандартів для розподілених обчислень, значний прогрес у швидкості обробки та ємності пам’яті, а також більш потужні, сумісні мови програмування. Надійна високошвидкісна мережа зменшує межі між віддаленими розробниками, такі структури програмування, як .Net і J2EE, забезпечують ефективний доступ до великої кількості інформації, пов’язаної з будь-яким проектом програмного забезпечення, а нові функції спільної роботи можна включати в IDE через відкриті інтерфейси прикладного програмування (API).

ПЗ охоплює широкий спектр завдань, починаючи від створення вимог до налагодження коду, і зараз дослідники починають розробляти прототипи інструментів ПЗ для будь-яких завдань. На даний час існують інструменти, які підтримують моделювання, проектування та керування програмним забезпеченням у реальному часі. Однак інструменти розробки зазвичай базуються на звичайних інструментах і технологіях. Наприклад, коли розробники реєструють або витягують код із сховища, користувачі можуть бути попереджені про можливі конфлікти. Існують інструменти для редагування та побудови діаграм у режимі реального часу, у яких звичайна модель копіювання, модифікації та об’єднання замінюється на повністю синхронний спільний доступ до файлів із підтримкою кількох переглядів.

Інструменти проектування. Інструменти проектування ПЗ частково або повністю зосереджені на підтримці співпраці під час проектування артефактів. Інструменти розробки CSE зосереджені на робочому процесі,

комунікації та базовій генерації вихідних файлів, а не на низькорівневому кодуванні. Інструменти цієї категорії зазвичай підтримують розробку відносно простих і малодеталізованих артефактів, таких як діаграми класу, послідовності та Class-ResponsibilityCollaborators (CRC).

Інші діаграми уніфікованої мови моделювання (UML), такі як діаграми переходів станів і діаграми варіантів використання, здаються надто складними для підтримки інструментами проектування CSE на даний момент, хоча нещодавно було випущено кілька комерційних реалізацій таких інструментів, як-от Poseidon для UML Enterprise Edition, представлений на рисунку 2.1. Poseidon підтримує спільне моделювання UML із блокуванням, якщо потрібно, і засобами виявлення та вирішення конфліктів. Крім того, хоча це не показано на поточному знімку екрана, Poseidon також підтримує спільну білу дошку та миттєвий обмін повідомленнями між розробниками.

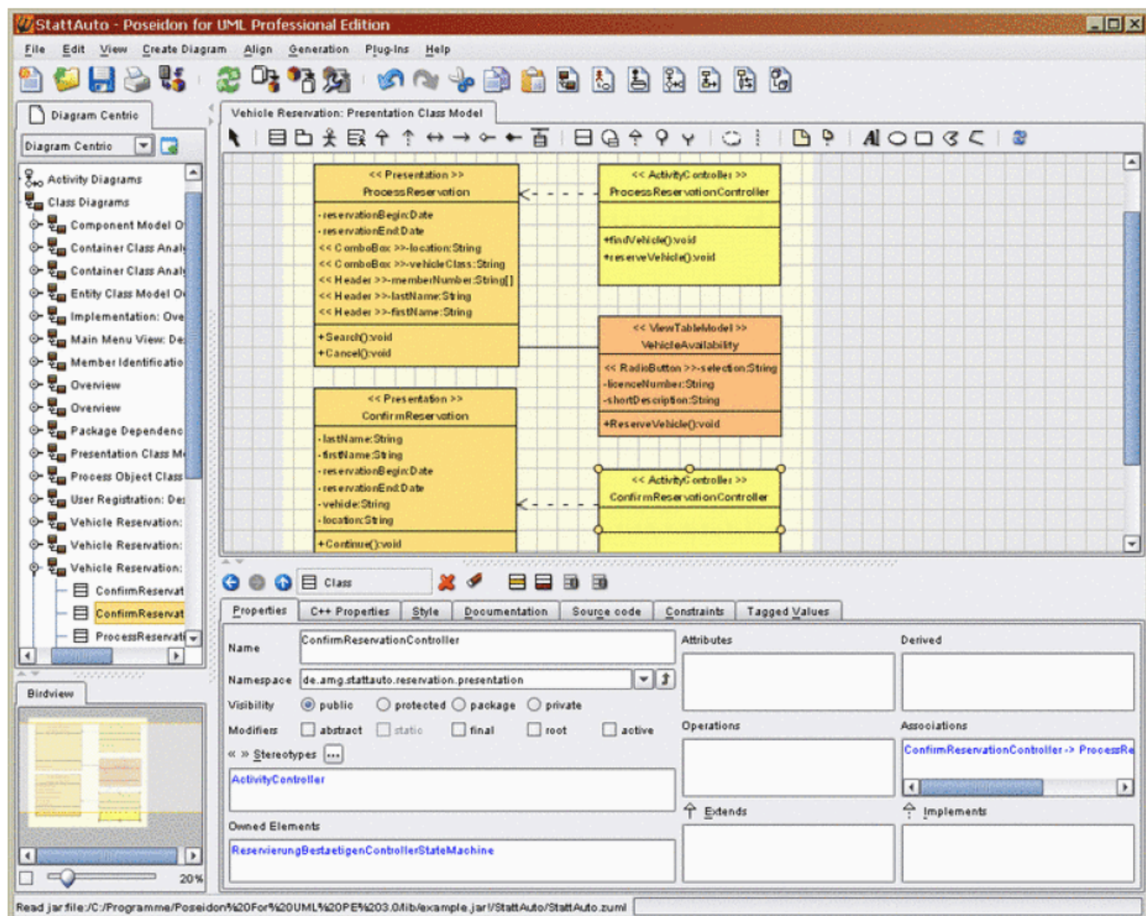


Рисунок 2.1 – Інструмент UML для спільної роботи

Rosetta [8] є відомим дослідницьким прототипом для веб-редагування спільного UML. Архітектура Rosetta дозволяє редагувати документи розробки програмного забезпечення на основі HTML з вбудованими діаграмами UML. Аплет редактора дозволяє спільно редагувати діаграми UML, як показано на рисунку 2.2. Rosetta також підтримує тести на відповідність коду, де вихідний код порівнюється з проектною документацією на наявність можливих невідповідностей.

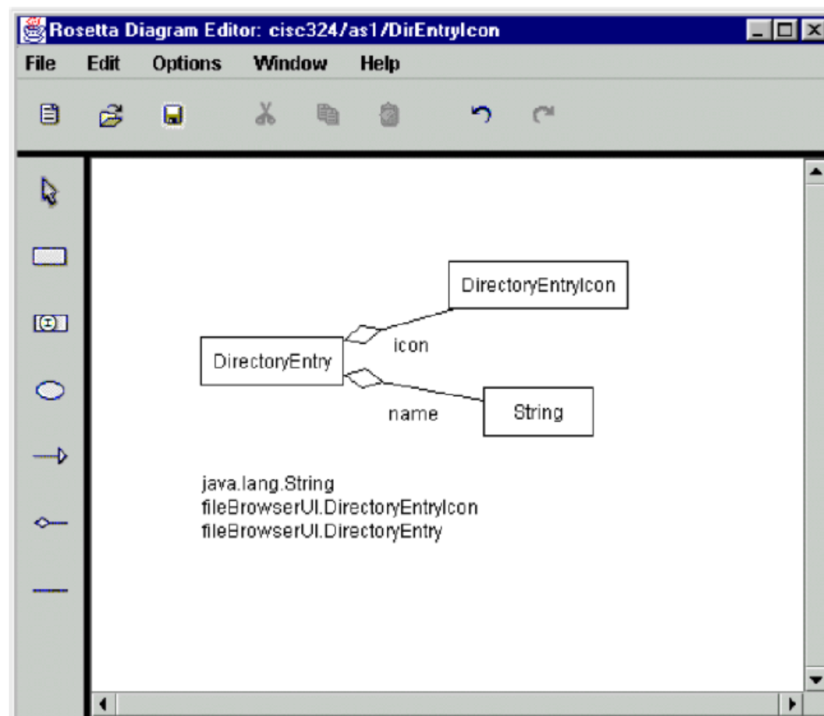


Рисунок 2.2 – Веб-інструмент для спільного проектування документів Rosetta

Інструменти розробки. Існує багато специфічних інструментів-прототипів для виконання ряду завдань розробки. Для розподіленого екстремального програмування була випущена система під назвою Moomba [9]. Середовище Moomba для розподіленого XP представлено на рисунку 2.3. Moomba полегшує повсякденну діяльність XP шляхом спільної роботи, де історіями користувачів та іншими артефактами XP можна ділитися та змінювати за допомогою кількох видів використання.

Moomba також підтримує повнофункціональну IDE для спільного спільного редагування, яка включає підсвічування синтаксису, доповнення коду, збірку та підтримку спільного налагодження.

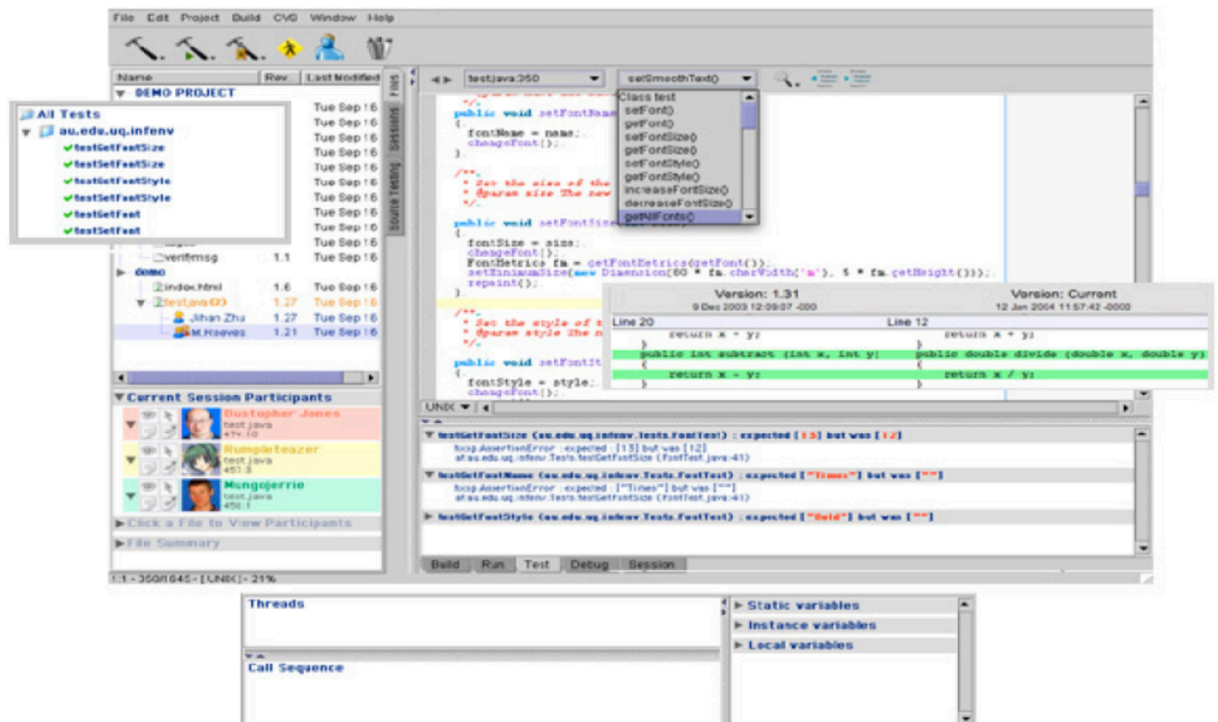


Рисунок 2.3 – Середовище спільної розробки XP Moomba

Moomba є наступником Tukan, інструменту CSE для редагування SmallTalk. Система Tukan представлена на рисунку 2.4. Tukan підтримує редагування вихідних файлів, але зміни коду не поширюються на інших користувачів, але замість цього Tukan забезпечує роботу у реальному часі про присутність інших користувачів та їхній вклади для внесення суперечливих змін. На рисунку 2.4 видно індикатори спільного коду Tukan, які передають інформацію про ступінь зацікавленості (DOI) та можливі проблеми конфігурації між програмістами.

За останній час багато великих комерційних IDE також зробили значні кроки до співпраці в режимі реального часу на рівні коду. З середовищ Java IDE майже всі підтримують спільні засоби розробки.

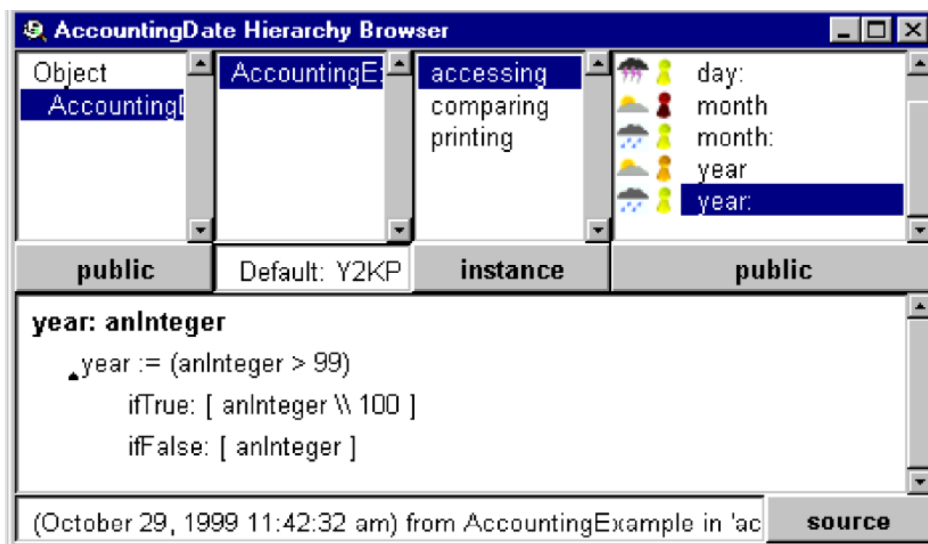


Рисунок 2.4 – Редактор коду для спільної роботи Tukan

Eclipse є найпопулярнішим середовищем розробки для Java, яке підтримується багатьма найбільшими корпораціями галузі.

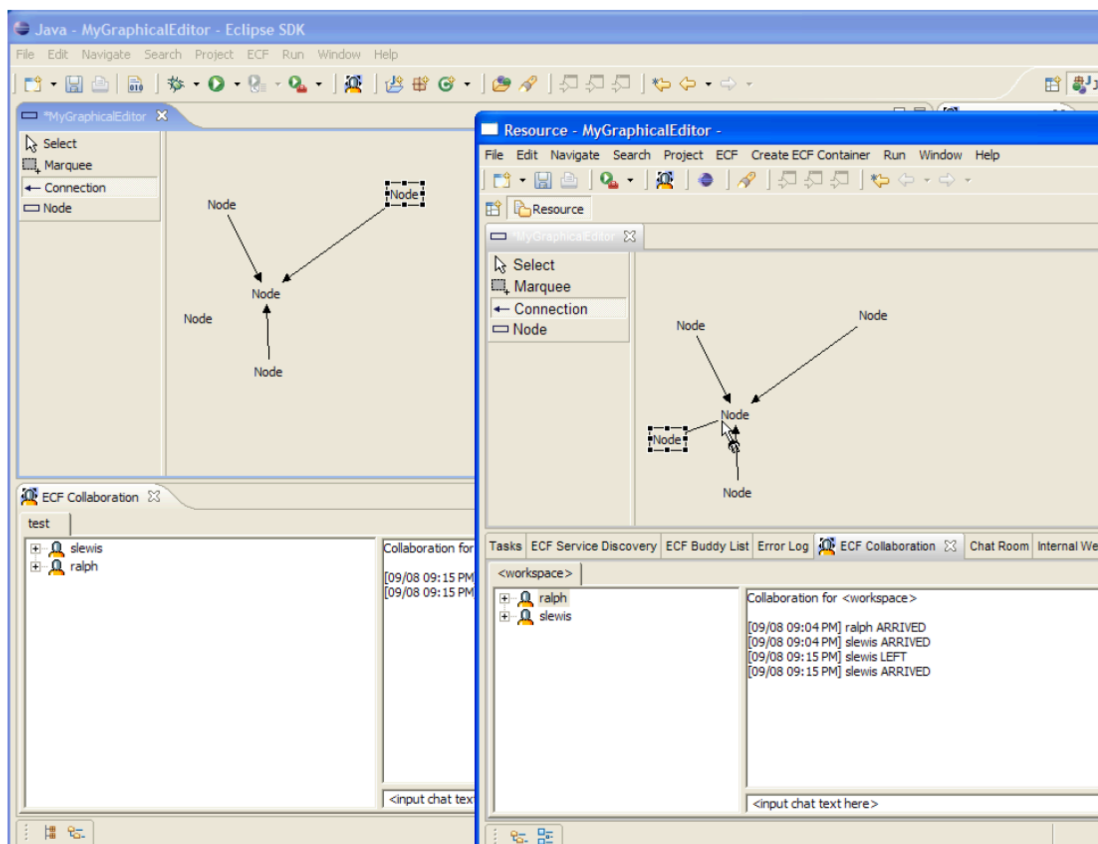


Рисунок 2.5 – Інструмент редагування графіка в Eclipse Communication Framework

Хоча сам Eclipse не підтримує співпрацю на рівні коду, але підпроект під назвою Eclipse Communication Framework має на меті використовувати сховище коду Eclipse і проектну модель спільно використовувати та редагувати спільно. Зараз доступний API для базового обміну, а також деякі прототипи клієнтських програм. Така програма представлена на рисунку 2.5, де спільний інструмент редагування графіків розміщено в IDE Eclipse.

Інструменти перевірки. Інструменти перевірки CSE зазвичай підтримують одну з двох функцій: дозволяють користувачам спільно перевіряти код і проекти як група або дозволяють окремим користувачам перевіряти код і дизайн, які були розроблені спільно.

Інструменти перевірки відрізняються від інструментів управління тим, що їх ключовою роллю є перевірка та дослідження артефактів для вдосконалення, на відміну від інструментів керування, які більше пов’язані з груповою координацією, проектуванням високого рівня та контролем артефактів.

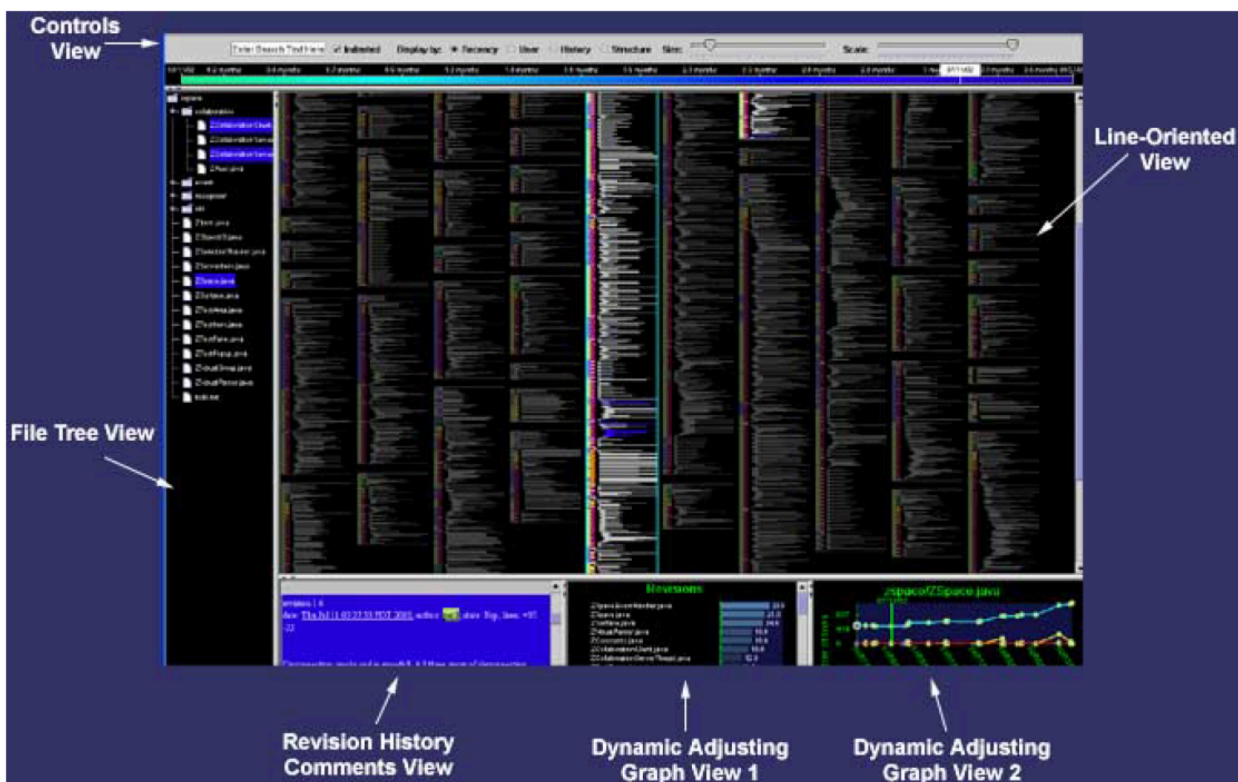


Рисунок 2.6 – Інструмент перевірки Augur

Прикладом популярного інструменту перевірки є Augur, інтерфейс якого показано на рисунку 2.6. Augur — це комплексний інструмент для перевірки та вивчення діяльності з розробки програмного забезпечення. Augur складається з архітектури збору даних на основі семантичного аналізу сховищ вихідного коду та набору інструментів візуалізації. Ці інструменти дозволяють розробникам відстежувати свою діяльність і досліджувати розподіл їхніх об’єднаних дій у часі та артефакти.

Для звітування про вплив змін існує архітектура Palantır. На рисунку 2.7 представлено компонент візуалізації Palantır, який інформує розробників про потенційно конфліктні вихідні файли із сховищ коду. Мета Palantır – підвищити обізнаність програмістів.

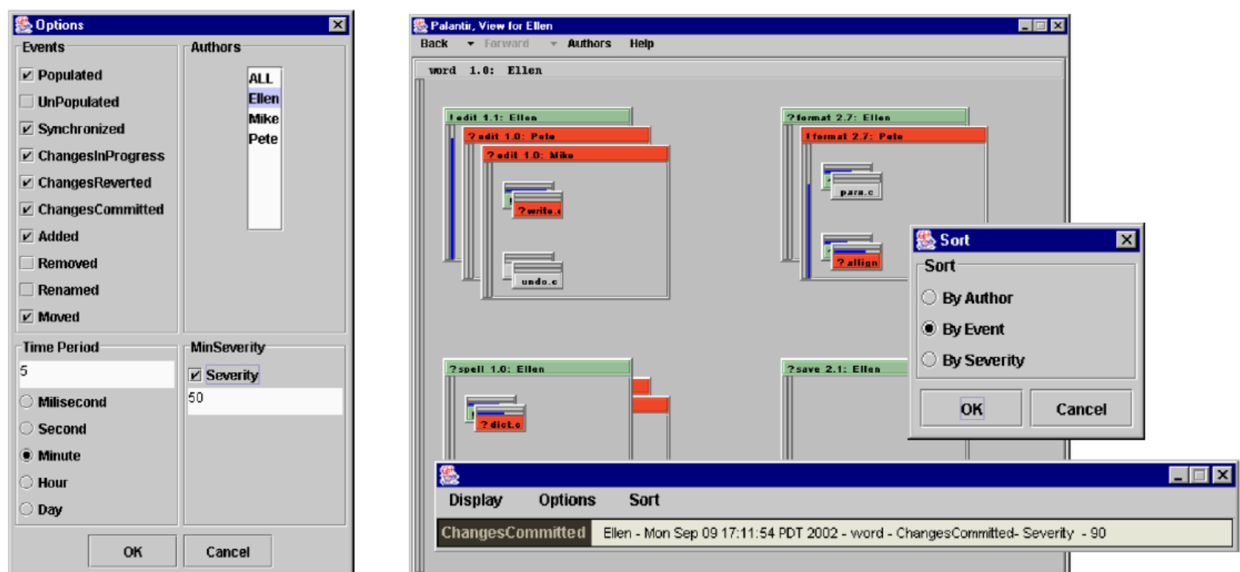


Рисунок 2.7 – Інструмент спільної візуалізації Palantır

Порівняння з інструментами на основі Caise. Перш ніж представити структуру Caise та пов’язані з нею інструменти, варто порівняти функції та можливості існуючих інструментів. Інструменти CSE, представлені раніше в цьому розділі, класифіковані в таблиці 2.1. Категорії, які використовуються для узагальнення цих інструментів, пояснюються наступному розділі.

З огляду на матрицю функцій, є очевидним, що інструменти CSE відрізняються за кількістю підтримуваних основних функцій. Це цілком очікувано, оскільки інструменти CSE адаптовані для конкретних цілей і, як правило, не потребують підтримки всіх завдань. Проте те, що вони створені для спільної роботи та написані без використання допоміжних фреймворків, гарантують високу вартість реалізації цих інструментів, але вони, як правило, не такі потужні, як універсальні однокористувацькі інструменти SE, такі як IDE.

У розробці структури Caise ключовою метою є можливість надати якомога більше основних функцій CSE для використання розробниками додатків. Замість того, щоб писати інструменти для всіх цілей CSE, має на меті забезпечити структуру для підтримки швидкого інструменту CSE. Враховуючи Caise, інструменти CSE мають потенціал для успішної підтримки всіх категорій, перелічених у наведеній вище матриці функцій, використовуючи сервіси інфраструктури.

Ключові відмінності між фреймворком Caise та іншими типами інструментів CSE полягають у тому, що інструменти на основі Caise легко розширюються за допомогою чітко визначених API, мають доступ до детальної інформації про проект через спільний семантичний аналізатор інкрементного вихідного коду, а інструменти є повністю синхронними, у яких будь-які кількість і типи інструментів можуть спільно редагувати артефакти в реальному часі, навіть з різних переглядів артефактів.

2.2 Приклад реалізації концепції шаблонів групової динаміки процесів розробки

Для підтримки корисних інструментів CSE важливо визначити вимоги, які програмісти висувають до таких інструментів. Щоб полегшити розробку інструментів, не менш важливо визначити основні функції, які знадобляться дослідникам під час створення нових типів інструментів. Дослідження, що

лежать в основі спільної структури Caise, базуються на моделях що будуть описані в у цьому розділі. Мета дослідження полягає в тому, щоб підтримати розробників, які працюють разом у спосіб, описаний шаблонами.

Шаблони зазвичай використовуються для документування повторюваних ситуацій і рішень. Концепції з архітектурної області є надзвичайно успішними, а шаблони проектування для ПЗ стали стандартною формою документації для створення програмних систем. Шаблони відрізняються ступенем деталізації, деякі описують прості концепції, такі як механізми для повторення списку, інші описують цілі організаційні структури. Успіх шаблонів у розробці програмного забезпечення привів до створення мов шаблонів для інших галузей.

Мова шаблонів. Шаблони проектування зазвичай описуються мовою шаблонів. Мова шаблонів намагається абстрактно визначити повторювані тенденції розробки програмного забезпечення. Хоча жодна мова шаблонів не отримала повної стандартизації, більшість описує наступний список властивостей:

– **Ім'я.** Загальна назва шаблону.

– **Контекст.** Загальна область, у якій можна застосувати шаблон.

– **Проблема.** Проблема, яку вирішує шаблон.

– **Фактори.** Фактори, які керують використанням шаблону для даного контексту. Включають легкість застосування шаблону, масштабованість дизайну та надійність.

– **Відомості.** Відомі небажані характеристики програмного забезпечення, які вказують на даний шаблон, можуть забезпечити належне полегшення.

– **Розв’язання.** Опис способу застосування поданого шаблону. Зазвичай це включає приклади кодування, діаграми UML і обговорення особливостей застосування рішення до конкретних контекстів.

– **Обґрунтування.** Обґрунтування того, чому шаблон бажаний і як рішення забезпечує кращий кінцевий дизайн, ніж інші підходи.

– **Приклади.** Огляд програмного забезпечення під час його рефакторингу відповідно до заданого шаблону. Приклади зазвичай обговорюють проблему, контекст і фактори, а також рішення.

– **Небезпечні місця.** Поширені підводні камені під час використання цього шаблону та рекомендовані обхідні шляхи.

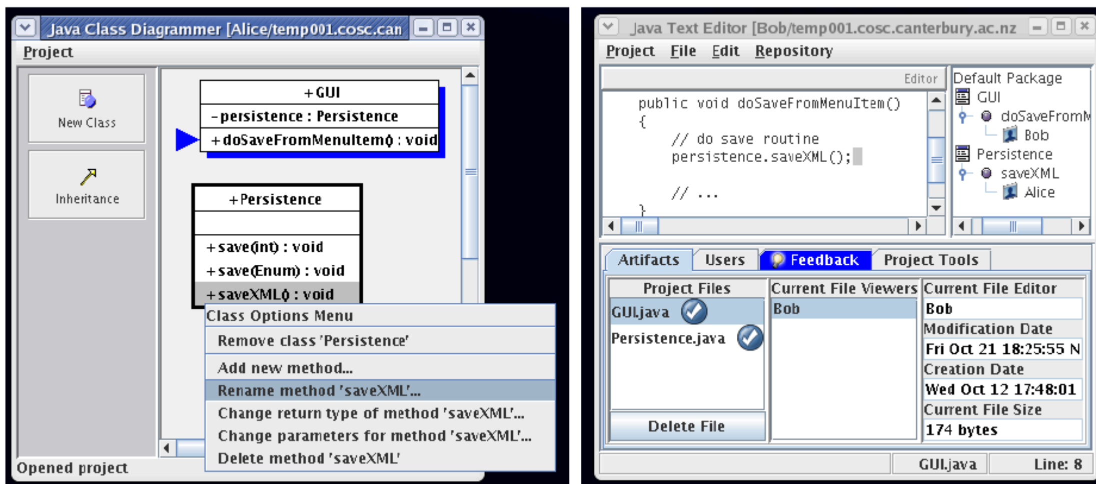
– **Відомі випадки використання**. Ідентифікація існуючих застосувань заданого шаблону в програмних системах або процесах.

– **Пов'язані шаблони.** Список спільних і пов'язаних шаблонів, а також шаблонів, які можуть надати альтернативне рішення

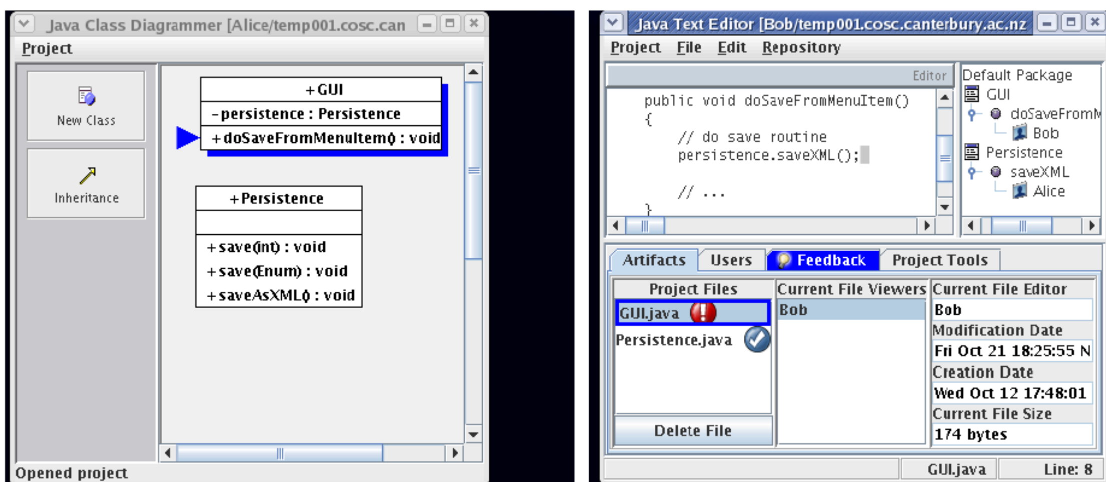
Відоме, як правило трьох, публікація шаблону його дизайнером зазвичай не рекомендується в спільноті шаблонів. Навпаки, третя сторона має визначити та задокументувати три незалежні приклади шаблону, який використовується в типових сценаріях SE. Це правило спрямоване на запобігання розповсюдженню слабких шаблонів.

Приклад шаблону. Розглянемо шаблон режиму розробки як шаблон-кандидат, визначений у полі CSE. Шаблон Mode of Development описує переважно способи взаємодії програмістів один з одним під час спільної роботи над певним набором артефактів. Існує багато режимів, один з яких це - «Дія/реакція». Режим розробки «Дія/реакція» інкапсулює повторювану поведінку такої типової ситуації: один програміст вносить зміни в кодову базу, другий програміст отримує сповіщення про можливий конфлікт, а потім обидва програмісти об'єднуються для вирішення конфлікту.

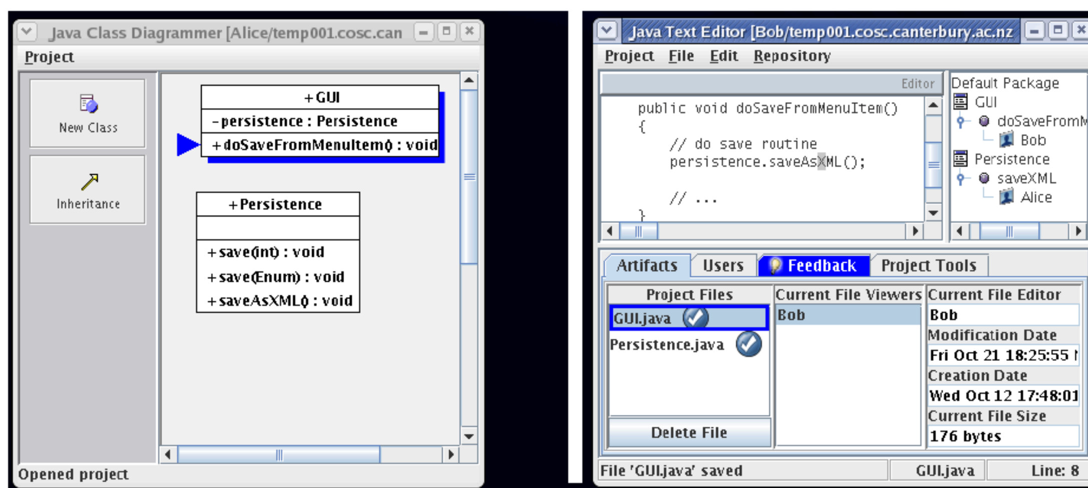
На рисунку 2.8 представлено шаблон «Дія/реакція». У цьому прикладі інструменти CSE на основі Caise використовуються для ілюстрації шаблону в контексті детальної взаємодії. Щоб створити сцену, користувач Аліса працює з інструментом діаграми класів, а користувач Боб працює з текстовим редактором. Цей приклад продовжує сценарій, представлений у першому розділі. Обидва інструменти працюють у режимі спільної роботи в реальному часі, працюючи на основі спільного коду.



а) Дія: перейменування методу



б) Повідомлення: проект став нестабільним



в) Реакція: Рефакторинг усіх відповідних викликів методів

Рисунок 2.8 – Приклад шаблону «Дія/реакція»

На рисунку 2.8 а) Аліса вирішує перейменувати метод saveXML() у класі Persistence на saveAsXML() через діаграму класу. Однак Аліса не знає, що користувач Боб нещодавно зробив новий виклик Persistence.saveXML() у файлі GUI.java і що перейменування методу порушить код.

На рисунку 2.8 б) Боб отримує повідомлення за допомогою механізму інформування про те, що проект нещодавно перейшов у неузгоджений стан. У нижній частині текстового редактора, представленому в правій частині рисунка 2.8 б), панель артефактів виділяє файл, який наразі містить семантичну помилку. Переглянувши панель зворотного зв’язку на цьому етапі, Боб буде проінформований про те, що метод, який він зараз редагує, здійснює виклик Persistence.saveXML() , який зараз не вирішено. На цьому етапі Алісу також буде повідомлено про ту саму проблему за допомогою механізмів зворотного зв’язку.

На рисунку 2.8 в) проблема вирішена. Використовуючи інформацію зворотного зв’язку, надану обом користувачам, або, можливо, просто розмовляючи один з одним, і Боб і Аліса можуть вирішити, що або операцію перейменування методу потрібно скасувати, або рефакторинг усіх викликів зміненого методу Persistence.saveAsXML() необхідно виконано. У цьому прикладі Боб просто оновлює виклик методу з файлу GUI.java і проект знову досягає стану, придатного для збірки. На цьому цикл подій «Дія/Реакція» завершується.

За наявності адекватної та відповідної інструментальної підтримки для шаблону «Дія/реакція» обидва користувачі будуть попереджені про наявну проблему, а також отримають можливість негайно проконсультуватися один з одним і виправити проблему.

Щоб описати шаблон «Дія/реакція» в термінах мови шаблонів, наведено кілька прикладів. Контекстом для цього шаблону є будь-яка ситуація, коли будь-яка кількість користувачів може вносити перекриваючі зміни у синхронних або асинхронних налаштуваннях. Проблему можна визначити як недостатню обізнаність, яка дозволяє вносити суперечливі

зміни без виявлення будь-якою стороною. Конкуруючі сили включають ступінь ізоляції, доступні механізми зворотного зв'язку та здатність розпізнавати суперечливі зміни. Потенційною небезпекою є тимчасові зміни, які можна безпечно ігнорувати, хоча ідентифікацію справді тимчасових змін, швидше за все, неможливо автоматизувати.

Без надійної інструментальної підтримки шаблону «Дія/реакція» Аліса та Боб могли б продовжувати спричиняти конфлікти. Якщо обидва користувачі не працюють тісно разом, вони, швидше за все, виявлять неузгоджений стан програми лише після синхронізації вихідних файлів. У цьому випадку обидва користувачі, ймовірно, спробують вирішити проблему, змінивши свої індивідуальні зміни, що знову порушить основну збірку проекту, коли обидва набори вихідних файлів буде зафіксовано назад у сховищі коду.

Слід зазначити, що навіть зі звичайними інструментами шаблон «Дія/реакція» все одно буде застосовуватися. Однак у таких випадках проміжок часу між кожною фазою шаблону буде довшим. Це пояснюється тим, що сповіщення про конфліктні дії зазвичай активується періодичними звітами про збірку з центрального сховища коду. Вирішенню конфлікту також можуть перешкодити інші модифікації, внесені до ідентифікації конфлікту.

При використанні більш складних інструментів, таких як IDE, співпраця існує навіть у проектах одного розробника. IDE зазвичай підтримують кілька переглядів артефактів, а це означає, що всі компоненти всередині IDE повинні співпрацювати один з одним, щоб підтримувати узгодженість своїх версій. Коли кілька розробників у команді використовують IDE як редактор коду, співпраця знову полегшується через сховище коду.

Незалежно від типів використовуваних інструментів, розробники зазвичай також використовують електронну пошту та інші способи комунікації, щоб допомагає їм координувати та спілкуватися на додатковому рівні, ніж це дозволяють їхні інструменти та сховище коду.

2.3 Побудова карт шаблонів для групової розробки

Деякі шаблони співпраці для IT проектів важко підтримувати звичайними інструментами, а це означає, що їх можна адаптувати лише з дуже неточною деталізацією. Наприклад, незалежні зміни вихідних файлів часто включають непередбачені побічні ефекти, такі як пошкоджені залежності коду. Наразі одним з способів виявлення конфліктів кодування між програмістами, які співпрацюють є інтеграція всього вилученого вихідного коду та дослідження помилок у створеному проекті. Це подовжує час розробки між випусками стабільних версій, а також може призвести до помилок при одночасних модифікаціях вихідного коду.

Можна передбачити, що з підтримкою інструментів у режимі реального часу для таких шаблонів, як незалежна модифікація коду, розробка програмного забезпечення може стати значно легшою.

Формальна ідентифікація шаблонів. Дослідження в цій роботі конкретно не передбачає формальної ідентифікації шаблонів CSE. Щоб розробити хороші інструменти CSE, можна моніторити процес, як інженери програмного забезпечення координують свої завдання та співпрацюють під час розробки.

Як обговорювалося вище, “правило трьох” стверджує, що розробники шаблонів не повинні бути тією ж стороною, яка розробляє шаблон. Для багатьох шаблонів існує багато повторюваних прикладів шаблонів CSE в рамках існуючих методів розробки програмного забезпечення та інструментів, що дає дослідникам можливість задокументувати та офіційно опублікувати такі шаблони, якщо це потрібно.

На рисунку 2.9 представлена карта шаблонів, орієнтована на CSE. На цій карті згруповані пов’язані сімейства шаблонів, описані в попередньому пункті, наприклад організаційні шаблони, шаблони групового програмного забезпечення та шаблони кооперативної взаємодії. Ця карта представляє повторювані тенденції взаємодії між співпрацюючими інженерами

програмного забезпечення і надається як засіб розуміння наслідків, конкуруючих сил і різних контекстів, пов'язаних з CSE.

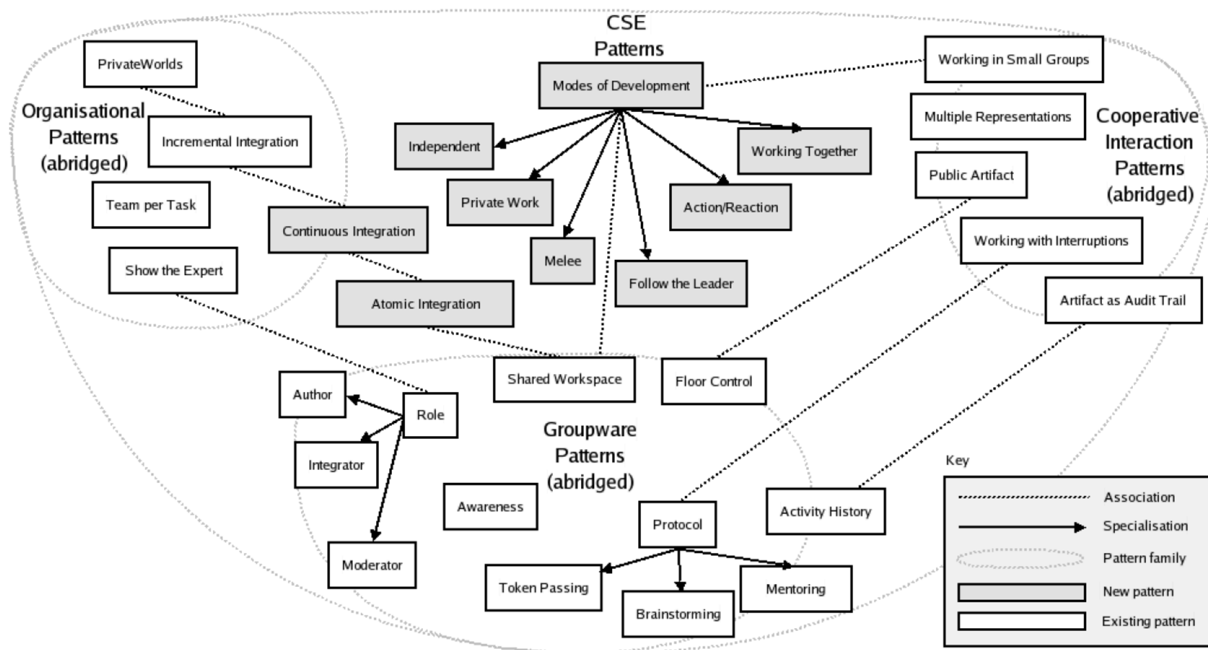


Рисунок 2.9 – Карта шаблонів CSE

CSE, по суті, є об'єднанням цих пов'язаних сімейств шаблонів з деякими додатковими специфічними характеристиками. Подібно до десяти шаблонів кооперативної взаємодії, проводиться робота з визначення цих особливих характеристик. На підставі досліджень [10 - 12], заснованих на випробуваннях прототипів інструментів, розглянемо два шаблони-кандидати для CSE: атомарна інтеграція та режими розробки.

Атомарна інтеграція. Є відомим шаблон інкрементної інтеграції [15], де модифіковані одиниці вихідного коду регулярно і часто повертається в основне сховище коду, щоб запобігти будь-яким серйозним помилкам у розробці. Більшість досліджень такої інтеграції стверджують, що достатньо щоденного процесу інтеграції коду. Згідно концепції безперервної інтеграції [16], постійна інтеграція заохочує розробників інтегрувати свої змінені вихідні файли відразу після будь-якої модифікації, а також повторно синхронізувати свій власний кеш незмінених файлів з останньою версією зі

сховища коду. Також існує деяка інструментальна підтримка безперервної інтеграції [17].

Повністю синхронні засоби розробки, такі як Poseidon і Moomba, дозволяють одночасно модифікувати спільні артефакти ПЗ. Оскільки артефакти обмінюються в режимі реального часу, жодних зусиль щодо інтеграції не потрібно. Даний режим керування конфігурацією визначається як атомарна інтеграція, де інтеграція насправді є постійною. Кожна модифікація миттєво включається в глобальний стан проекту з відповідним оновленням усіх змін.

Режими розвитку. Шаблон протоколу для CSCW було попередньо визначено в колекції шаблонів групового програмного забезпечення. Режими роботи, визначені шаблоном протоколу, такі як передача маркерів і мозковий штурм, добре підходять для загального CSCW. Для CSE, однак, зустрічаються більш специфічні режими роботи. Було визначено кілька режимів взаємодії [18], кожен з яких характеризується ступенем необхідної координації та характером діяльності:

– **Приватно:** користувач фактично тимчасово виходить із групи, щоб переконатися, що зміни були успішними, перш ніж надавати доступ іншим членам команди. Такий користувач може вимагати, щоб решта проекту виглядала замороженою в часі, але в ідеалі має бути можливість інтегрувати зміни, а не повторювати їх публічно.

Атомарною одиницею взаємодії для приватної роботи є набір вихідних файлів після виконання приватної роботи модифікації які об’єднуються в основну версію проекту. Повідомлення про зміни у випадку приватної роботи, будуть відкладені до періоду інтеграції коду. Проте все ще можна попередити всі сторони про суперечливі зміни, навіть якщо копія вихідних файлів проекту розробляється приватно. Ось як працює інструмент Tukan CSE.

Прикладом приватної роботи є розробка користувачем складного алгоритму, який мало пов’язаний з рештою проекту. У цьому випадку

розробник може віддати перевагу працювати в повній ізоляції, знаючи, що події від інших пов'язаних користувачів, будуть не суттєвими. Після впровадження алгоритму та інтеграції з основним проектом розробник повернеться до більш спільного режиму розробки. Приватна робота є ключовим шаблоном, який використовується в традиційній інтеграції артефактів ПЗ за допомогою ідіом копіювати/змінювати/об'єднувати.

– **Незалежні (Independent) користувачі:** користувачі розташовані в частинах коду, семантичні зв'язки яких досить слабкі, щоб вони могли безпечно вважати незалежними. Немає необхідності в частому спілкуванні, а незалежні оновлення не загрожують цілісності проекту.

Атомарна одиниця взаємодії для самостійної роботи може бути найбільш корисною на рівні семантичної зміни. У цьому випадку, коли область коду була суттєво змінена, наприклад, перейменовано метод або оголошено додатковий клас, інші пов'язані користувачі будуть повідомлені. У випадку незалежної розробки малоймовірно мати перекриваючі модифікації, які викликають конфлікти.

Прикладом самостійної роботи може бути редагування користувачем класу A графічного інтерфейсу для зміни меню, редагування користувачем B класу запису клієнта (моделі), і додавання користувачем C нового пакета, який ще не взаємодіє з іншими класами. У цьому випадку ймовірний лише незначний зворотній зв'язок між користувачами, а спілкування буде на низькому рівні. Шаблон кандидата незалежного режиму роботи зазвичай демонструється у великих, добре скоординованих проектах програмного забезпечення.

– **Слідкуйте за лідером (Follow the Leader):** один користувач є основним і може вносити зміни в код в режимі реального часу. Щоб узгодити перегляди, можна використовувати строгий метод WYSIWIS, особливо якщо всі користувачі використовують один інструмент.

Атомарна одиниця взаємодії для ситуації лідера має бути детальною. В ідеалі модифікація будь-якого типу має бути негайно поширена від лідера до

всіх членів команди. Оскільки лідер є єдиною людиною, яка вносить зміни, контроль порядку подій є тривіальним. У середовищах із суворим WYSIWIS інструменти CSE поширюватимуть зміни у поглядах керівника на всі інші інструменти.

Даний сценарій може складатися з того, що один ключовий розробник показує деталі нещодавно завершеної зміни. Іншим прикладом може бути показ групі розробників, як змінити один із кількох класів, які потребують одного типу рефакторингу. Шаблон кандидата-лідера демонструється в JBuilder IDE за допомогою механізму спільного використання файлів на основі маркерів.

– **Концепція "Спільна робота".** Працюючи разом члени команди перевіряють і редагують проект як пакет і в деяких ситуаціях це може означати що група з двох або трьох розробників які працюють над однією фізичною областю коду. В інших ситуаціях це може означати, що група розробників вносить обережні та продумані зміни до областей дуже пов'язаних областей коду. Цей спосіб розробки дуже схожий на «Слідуй за лідером», за винятком того, що всі члени групи, швидше за все, будуть залучені до модифікації проекту, а не лише лідер.

Оскільки розробники в цьому режимі дуже тісно співпрацюють, потрібна тонка одиниця взаємодії, яка дозволяє негайно поширювати всі зміни. У випадку редакторів вихідного коду це може означати поширення змін на основі кожного символу. Соціальні протоколи, ймовірно, диктують порядок подій під час спільної роботи; наприклад, малоймовірно, що один користувач вибере та видалить весь метод, якщо він або вона знає, що інший користувач зараз його змінює.

Прикладом спільної роботи може бути спроба трьох користувачів розділити великий клас на два менших. Один користувач може визначити другий клас, помістивши його у відповідний пакет. Другий користувач може почати переміщення відповідних методів з першого класу до другого класу. Третій користувач може почати пошук нещодавно зламаних посилань і

розпочати їх виправлення. До завершення цього завдання дуже ймовірно, що користувачі будуть часто спілкуватися, координувати свої зусилля та обговорювати наслідки дизайну.

– **Дія/реакція:** існують сильніші обмеження, оскільки користувачі стають ближчими у логічному чи семантичному плані. Зміни, внесені користувачем (дії) до таких аспектів, як кількість і тип властивостей класу, параметри і повертаються типи методів або успадкування та структура інтерфейсу вимагатимуть відповідей (реакцій) від інших користувачів, на роботу яких потенційно може вплинути дана дія. Механізми інформування можуть попередити користувачів про можливі загрози (наприклад, інший користувач редагує суперклас). Потім для обговорення та вирішення проблем можна використовувати інші механізми підтримки співпраці.

Атомарна одиниця взаємодії для режимів дії/реакції має бути відносно невеликою. На найнижчому рівні кожна подія, яка оновлює семантику проекту, повинна бути поширена для перегляду всім іншим користувачам-учасникам. Незважаючи на те, що більшість поширених змін, залишаться непоміченими, оскільки вони не впливають безпосередньо на роботу інших користувачів, у той момент, коли модифікація викликає конфлікт для іншого користувача, має відбутися сповіщення.

Прикладом може бути один користувач, який змінює тип параметра у визначенні методу в класі C1. Іншому користувачеві, який редагує клас C2, може знадобитися оновити щойно створені виклики цього методу. Чим швидше дія стане доступною для всіх пов’язаних користувачів, тим краще з точки зору уникнення плутанини та затримок розробки.

– **Концепція “Melee”:** Кілька користувачів вносять (потенційно) суперечливі зміни в набір артефактів і вони будуть у стані зміни протягом певного періоду. Такі зміни, як правило, обговорюються заздалегідь і опосередковуються функціями інфраструктури, як чат.

Немає очевидної атомарної одиниці взаємодії для для способу розвитку. Для груп, де комунікація обмежена, наприклад, у розподіленій розробці,

дрібні зміни, такі як модифікації кожного символу вимагатимуть виявлення та поширення серед усіх користувачів, які співпрацюють. В інших ситуаціях атомарна одиниця взаємодії може бути лише відносно грубою, щоб зменшити постійні переривання. Рівень деталізації змін зрештою залежить від існуючих соціальних протоколів для способів розвитку.

Прикладом режиму розробки melee може бути велика рефакторингова робота, коли всі розробники знають, що рефакторинг виконується. У цьому випадку певні обов'язки з ре факторингу були розподілені заздалегідь. Протягом цього періоду розробки відгуки про стосунки між користувачами-учасниками та залежностями коду, що наразі зламано, можуть бути замінені розширенішим механізмом, таким як аудіоконференції. Режим розробки можна спостерігати в більшості звичайних проектів розробки програмного забезпечення, коли відбувається період узгодженого рефакторингу з використанням звичайних інструментів контролю вихідного коду.

На базовому рівні ці способи розвитку очевидні в сьогоднішніх практиках, навіть без підтримки інструментів CSE. Враховуючи прогрес у бік більш синхронної підтримки інструментів для загалом спільних завдань, також можливо, що ці режими розробки забезпечують відповідний підсумок основних шаблонів взаємодії для всіх сфер комп'ютерно-опосередкованої взаємодії.

Застосування шаблонів спільної розробки програмного забезпечення. Щоб допомогти дослідникам у розробці нових інструментів CSE, вони повинні знати про закономірності, пов'язані з CSE. Ці шаблони представляють повторювані теми розробки програмного забезпечення, які мають складні особливості проектування та впровадження. Карта шаблонів CSE, представлена на рисунку 2.9 є відправною точкою при розгляді дизайну будь-якого інструменту CSE.

Посилаючись на цю карту шаблонів CSE, розробники інструментів CSE повинні брати до уваги режими розробки, які підтримуватимуть інструменти.

У той час як потужний інструмент CSE може підтримувати всі режими розробки, інші інструменти можуть підтримувати лише один режим. У цьому випадку фундаментальний дизайн інструментів CSE, ймовірно, буде відрізнятися. Наприклад, якщо режим «Слідувати за лідером» є єдиним підтримуваним режимом, тоді суворий WYSIWIS може бути єдиним переглядом, який вимагає впровадження.

Тип інструменту, що розробляється, також вносить особливі міркування. Для спільного інструменту діаграми класів, можливо, передбачається незалежний режим розробки. У цьому випадку блокування кожного поточного зміненого розділу семантичної моделі проекту може бути реалізовано для контролю паралелізму.

Для інструменту CSE, який підтримує як вихідний код, так і діаграми, варто дослідити шаблон кількох представлень. Для режиму інтеграції необхідно прийняти рішення щодо підтримуваних рівнів деталізації співпраці. Розробник інструменту повинен визначити, чи потрібно вносити та інтегрувати модифікації поступово, безперервно чи атомарно. У деяких інструментах можна підтримувати кілька рівнів деталізації співпраці.

Ще один важливий аспект — приватні області. Якщо потрібно підтримувати приватні області, дозволяючи розробникам працювати ізольовано, необхідно враховувати низку ключових аспектів:

– Скільки часу розробник може працювати в приватному робочому просторі?

– Чи обмеження залежить від часу чи відчутних зусиль інтеграції?

– Як підтримуватиметься повернення до основного спільного проекту?

– Чи будуть механізми поінформованості надані користувачеві, коли він або вона працює над окремою кодовою базою?

– З точки зору ролей розробника, чи будуть певні ролі для різних користувачів, чи це обробляється на вищому рівні?

Більшість інструментів CSE, які існують сьогодні, побудовані навколо конкретних процесів SE. Poseidon [11], наприклад, будує свою підтримку в основному на розробці програмного забезпечення. Moomba [32] підтримує розробку на основі парного програмування. Інструменти для одного користувача також часто підтримують певні ролі. Таким чином, ще одна міркування полягає в тому, які ролі підтримуватимуться інструментом CSE, чи лише соціальні протоколи забезпечуватимуть адекватне керування взаємодією користувачів.

Присутність користувача є ще одним дуже важливим аспектом для будь-якого інструменту CSE. Шаблони групового програмного забезпечення забезпечують відправну точку для забезпечення адекватного дизайну системи з точки зору обізнаності користувачів і зворотного зв'язку [33].

Модель кооперативної взаємодії «Артефакт як журнал аудиту» має цікаві наслідки для CSE. Більшість систем спільного редагування на основі CSCW працюють із тимчасовими артефактами, такими як спільні дошки та історія переглядів. Однак для CSE у більшості випадків історія модифікацій артефактів є корисною функцією інструменту. Якщо записується історія, чи обмежується вона модифікаціями артефактів? Або слід реєструвати додаткову інформацію, таку як взаємодія користувача, спроби створення проекту та семантичні події, такі як додавання нового класу чи вирішення посилання?

Як показано в цьому розділі, є кілька серйозних проблем, які слід враховувати під час розробки інструментів для підтримки CSE. Карта шаблонів для CSE є корисною для визначення системних вимог. Після визначення основних вимог інструменту CSE посилання на відносні шаблони, потрібні на етапах проектування та розробки інструменту.

Антишаблони співпраці. Програмні антишаблони [17] — це повторювані теми розробки або дизайну, які негативно впливають на процес SE. Одним із прикладів загального антишаблону є “God Object”, де об’єкт має занадто багато знань про всі інші об’єкти в системі. Це порушує

загальноприйняті принципи програмування, такі як інкапсуляція даних і низький зв'язок.

У контексті CSE деякі загальноприйняті шаблони для сприяння процесу SE можуть призвести до проблем за певних обставин. Практики SE базуються на існуючій підтримці інструментів. Передбачається, що як тільки інструменти CSE стануть звичайними для групового розвитку, деякі прийняті на даний момент ідіоми SE можуть застаріти або бути заміненими. Сюди входять обмеження розробки в парах лише двома розробниками та організаційна схема «Особи перед віддаленою роботою» .

Шаблони, які раніше вважалися хорошою практикою, можуть бути врешті-решт перекласифіковані як антишаблони SE в деяких сценаріях програмування. Однак іноді будуть потрібні шаблони, ортогональні CSE. Наприклад, кілька команд можуть час від часу працювати над окремими базами коду та інтегрувати свої зміни назад в основний проект, незважаючи на здатність інструментів CSE забезпечити повністю синхронну інтеграцію.

Шаблон Private Worlds. Шаблон Private Worlds детально досліджується, щоб надати приклад того, як наразі технологія SE може обмежувати практику програмування.

Безсумнівно, є час і місце для розробки «Private Worlds», використовуючи сховища коду для інтеграції зусиль офлайнової розробки назад у основну гілку проекту. Концепція приватної роботи також була визначена як спосіб розробки, який має підтримуватися інструментами CSE. Однак проблема полягає в тому, що приватна робота є переважно єдиним способом розробки для інженерів програмного забезпечення на даний момент.

Наступні результати дослідження надають аргументи проти постійного захисту від «змінних залежностей розвитку»:

1. Надто багато часу витрачається на виправлення помилок через обмежене спілкування [15]. Навіть програмісти, які працюють «приватно», щодня витрачають до половини свого часу на співпрацю, а не на кодування;

2. Чим раніше виявляються конфлікти, тим раніше вони вирішуються. Крім того, чим більше часу потрібно для вирішення проблеми, тим дорожчим стає програмний проект;

3. Програмуванням із використанням приватних робочих просторів важко керувати, новим користувачам важко отримати визнання, а час виходу на ринок йде повільно;

4. Інструменти об’єднання все ще мають проблеми з одночасно редагованими вихідними файлами ;

5. Концепція приватної розробки для окремих програмістів погано масштабується [16]. Закон Брукса стверджує, що «складність і комунікаційні витрати проекту зростають із квадратом кількості розробників» [38].

6. Репозиторії коду нового покоління починають змінювати конвенції SE на основі сховищ. Вони підтримують більш синхронне розповсюдження коду завдяки частому оновленню всіх кодових баз розробників за допомогою тонких одиниць зміни.

Шаблон Private Worlds безперечно виправданий у випадках, коли ймовірні низькі зусилля з інтеграції коду. Однак, що приватні робочі зони є основними засобами для звичайної розробки програмного забезпечення, навіть у командах, де заохочується регулярна та часта співпраця.

Тому шаблон класифікують Private Worlds як один із антишаблонів CSE, якщо використовувати його недоцільно. Це пов’язано насамперед з обмеженнями, які приватні робочі зони накладають на спілкування та обізнаність користувачів, а також з великими зусиллями, які часто потрібні для інтеграції коду назад в основний проект.

Висновки до розділу 2

В даному розділі досліджено інструменти групової розробки, визначено категорії таких інструментів, представлена матриця функцій, яка порівнює ці існуючі інструменти CSE з інструментами на основі Caise. У розділі

визначено моделі співпраці, очевидні в рамках SE. Виявляючи повторювані режими роботи між розробниками в рамках спільних проектів програмного забезпечення, інструменти CSE можна спроектувати відповідно до спостережуваних ключових вимог розробників, які співпрацюють.

Також досліджена концепція шаблонів CSE. Ці шаблони описані з метою мотивації, надаючи приклади типових сценаріїв. Шаблони, представлені в цьому розділі, не вважаються повними. Мета надання цих прикладів шаблонів полягає в тому, щоб проілюструвати основні ситуації, які повинні підтримувати інструменти CSE; нездатність підтримувати ці різні режими роботи свідчить про необхідність подальшої роботи над розробкою інструменту CSE. Деякі шаблони традиційної розробки програмного забезпечення були ідентифіковані як потенційно шкідливі, якщо використовувати їх у повністю спільних системах.

РОЗДІЛ 3. ПІДВИЩЕННЯ СЕМАНТИЧНОЇ ЕФЕКТИВНОСТІ МОДЕЛЕЙ ШЛЯХОМ РОЗРОБКИ ФРЕЙМВОРКУ СЕРЕДОВИЩА КОМАНДНОЇ РОЗРОБКИ

3.1 Розробка програмного забезпечення на основі семантичної моделі

Для належної підтримки шаблонів CSE, які спостерігаються під час командної розробки, інструменти потребують не лише вихідних файлів, щоб надати повну інформацію про створення програми. У той час як вихідні файли є важливим засобом введення для програмного продукту, технологія та обчислювальна потужність, доступні сьогодні, дозволяють набагато більш детальний аналіз програмного забезпечення. Подібним чином вихідний код не є ідеальним засобом обміну інформацією для підтримки складної взаємодії між кількома інструментами.

Вихідний код повний неявних зв’язків, які вимагають ретельного семантичного аналізу, щоб перетворити послідовність символів у корисну інформацію SE. Потужні IDE майже напевно вимагають внутрішньої побудови семантичної моделі проекту для аналізу змін коду та забезпечення таких функцій, як доповнення коду та перегляд ієрархії класів. Наприклад, Eclipse, Netbeans і Together Architect [46] мають вбудовані комплексні семантичні аналізатори для створення повної моделі програмного забезпечення проекту.

Таблиці символів [1] є найпростішим типом семантичної моделі; вони використовуються переважно компіляторами для перетворення абстрактних синтаксичних дерев (AST) у машинний або байтовий код. У той час як таблиці символів є точним джерелом інформації для компіляторів, більш багаті типи інформації потрібні для інструментів, які підтримують автоматичний рефакторинг, аналіз показників і запити до бази коду.

Повна семантична модель проекту програмного забезпечення, де доступна велика кількість типів інформації, як-от зв'язки між усіма оголошеннями програми, є надзвичайно корисним активом для більшості інструментів SE. Типова семантична модель програмного забезпечення містить представлення всіх оголошених сутностей, таких як класи та методи, разом із картою всіх зв'язків, таких як усі підкласи для даного суперкласу та всі виклики методів даної декларації методу. Щоб допомогти проілюструвати, що охоплює семантична модель, спрощений приклад семантичної моделі представлено на рисунку 3.1.

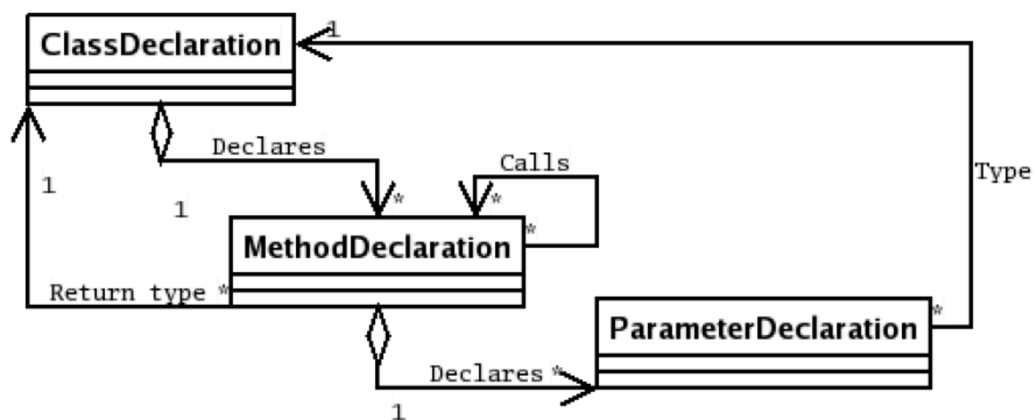


Рисунок 3.1 – Діаграма класів UML для спрощеної семантичної моделі програмного забезпечення

Щоб повністю змоделювати програмне забезпечення, відповідний проект семантичної моделі міститиме велику кількість класів і зв'язків. Семантична модель для Java, наприклад, представлена в цьому розділі. Ця модель, здатна описати будь-який компілюваний набір вихідних файлів Java, містить приблизно тридцять основних класів і понад сотню зв'язків між ними.

Для повної підтримки CSE необхідін інструменти, які працюють на семантичних моделях проектів. Інформацію про програму можна отримати за допомогою евристичних підходів, таких як зіставлення шаблонів токенів, але для програмного забезпечення лише цей тип синтаксичного аналізу може

дати неправильні результати. Лише за допомогою повного семантичного моделювання інструменти можуть впевнено визначити зв'язки між одночасними модифікаціями та вплив змін, що очікують на розгляд.

Техніка використання спільної семантичної моделі між розробниками суттєво відрізняється від усіх інших підходів, про які я знаю. Наприклад, незважаючи на те, що IDE використовують семантичну модель для надання багатих функціональних можливостей кожному користувачеві, вони все одно повертаються до систем сховища коду на основі файлів, щоб забезпечити поширення змін між розробниками. Не робиться жодних спроб поінформувати пари розробників про частки пов'язаного коду, що збігаються, або про вплив локальних модифікацій на найновішу версію проекту.

Враховуючи можливість атомарної інтеграції змін коду в міру їх виникнення, як обговорювалося в попередньому розділі і здатність перекладати модифікації артефактів у переклади семантичної моделі, то усі користувачі ефективно працюють над єдиним екземпляром проекту в реальному часі. Завдяки цьому з'являється можливість негайно виявляти сфери інтересів, які є спільними для групи користувачів під час навігації програмним забезпеченням, що розробляється.

Приклад спільної сфери інтересів у програмному забезпеченні представлено на рисунку 3.2. У цьому прикладі користувач Carl редагує метод під назвою update() у класі AnimatedSprite. У той же час користувач Wal редагує властивості в класі DynamicSprite, який є суперкласом AnimatedSprite. Суперклас класу Wal називається Sprite , який наразі не має оголошеного суперкласу.

З наявністю або без наявності інструментів, які працюють на спільній семантичній моделі програмного забезпечення проекту, очевидно, що на цьому етапі існує збіг між Carl та Wal. Обидва користувачі повинні тісно координувати свої дії, оскільки будь-яка зміна, яку вносить Wal може мати значний вплив на клас, який редагує Carl. Наприклад, якщо Wal змінює будь-які властивості в DynamicSprite, ці зміни негайно успадковуються

AnimatedSprite через семантику мов ООП. Подібним чином, якщо Wal оголошує метод під назвою update() у суперкласі DynamicSprite, це може змінити кількість звернень до методу під назвою update(), над яким зараз працює Carl.

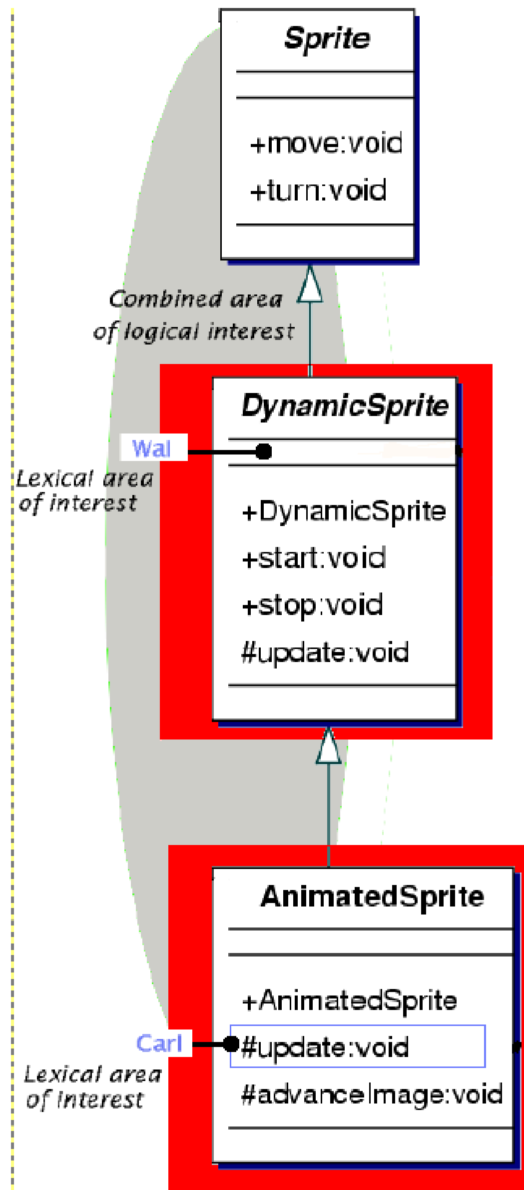


Рисунок 3.2 – Комбіноване середовище для двох розробників, використовуючи нотацію UML

Відповідним терміном для цієї області пов’язаного коду є сусідство в коді. Околиці коду — це вся область коду, семантично пов’язана з поточною

зоною фокусування користувача. Це також можна розглядати як область дії для будь-якої даної точки в проекті програмного забезпечення, беручи до уваги лексичний обсяг, успадкування, виклики методів, композицію та всі інші ідентифіковані семантичні відносини в структурі програмного забезпечення проекту.

Якщо існує підтримка в режимі реального часу для спільної семантичної моделі програмного забезпечення проекту, ідентифікація околиці коду для будь-якого даного користувача може бути обчислена негайно, коли користувач переходить від однієї частини коду до іншої. Без атомарної інтеграції змін можна обчислити лише околиці коду для останньої зафіксованої версії вихідних файлів проекту.

Враховуючи те, що оточення для будь-якого користувача можна обчислити в реальному часі для інструментів CSE, які використовують спільну семантичну модель, потрібне лише просте обчислення, щоб визначити, чи знаходяться два користувачі в межах пов’язаної області коду, або, іншими словами, семантично пов'язані між собою. Це дуже важлива перевага інструментів CSE у реальному часі перед їхніми звичайними аналогами: розробники можуть бути попереджені про збігаються сфери інтересів негайно, а не під час роздумів під час вирішення конфлікту у відповідь на невдалу реєстрацію сховища або створення.

Використання сусідства коду. Наслідки автоматичного розрахунку околиць коду та перевірки областей інтересу, що перекриваються, є великими. Як показує приклад, представлений на рисунку 3.2, для повного розуміння критичних областей, пов’язаних з будь-яким одним рядком коду, інструменти повинні знати про логічну структуру проекту, а не лише про оголошення, що містяться в лексичній області. Насправді інструментам може знадобитися шукати набагато далі, ніж безпосередня логічна структура, така як ієрархія успадкування; Інструменти також часто потребують визначення того, які інші частини системи залежать від основних класів у фокусі, і, у свою чергу, які області проекту ці класи залежать від них самих.

У програмному забезпеченні орієнтованого орієнтування є багато неявних і тонких, але важливих взаємозв'язків, які потрібно визначити та зрозуміти. Навіть найдосвідченіші групи розробників час від часу вносять неправильні зміни в проект, тому що тонка залежність між одиницями коду була пропущена. Ось чому інструментальна підтримка спільного кодового середовища є важливою — розробникам не обов'язково зберігати уявну картину всієї семантичної моделі, правил мови та поточного розташування всіх інших користувачів; Інструменти CSE мають потенціал для проактивного надання контекстно-специфічної інформації про відповідні області коду.

Було б дуже важко передбачити конфліктні модифікації між пов'язаними областями коду до того, як вони відбудуться, але інструменти CSE можуть сповіщати користувачів про семантичну близькість інших у реальному часі. Це помітна різниця між звичайними інструментами на основі репозиторію та інструментами CSE на основі семантичної моделі реального часу.

3.2 Архітектура фреймворку для спільної командної розробки програмного забезпечення

Мета інструментів CSE на основі Caise (Collaborative Architecture for Iterative Software Engineering) — дозволити програмістам працювати разом, не зменшуючи при цьому спілкування. Спілкування важливе для уникнення конфліктів кодування, обміну ідеями та вирішення проблем. Caise досягає цього, підтримуючи всіх програмістів синхронізованими в режимі реального часу і водночас надаючи обізнаність користувача та інформацію про стан проекту окремим інструментам. Інструменти на основі Caise підтримують те, що не надають сховища коду: спілкування між розробниками та інструментами під час детальної співпраці в реальному часі.

Для роботи було використано зовсім інший засіб для полегшення створення інструменту CSE у формі повністю синхронного підходу.

Загальний схематичний вигляд такої структури Caise , представлений на рисунку 3.3. Ключовими поняттями фреймворку є спільний набір артефактів, які окремі інструменти можуть редагувати в режимі реального часу і сервер, який координує дії кожного користувача та інструменту. За допомогою структури різні типи інструментів CSE можуть працювати разом у спільному проекті в режимі реального часу. Архітектурні деталі структури Caise представлені в даному розділі.

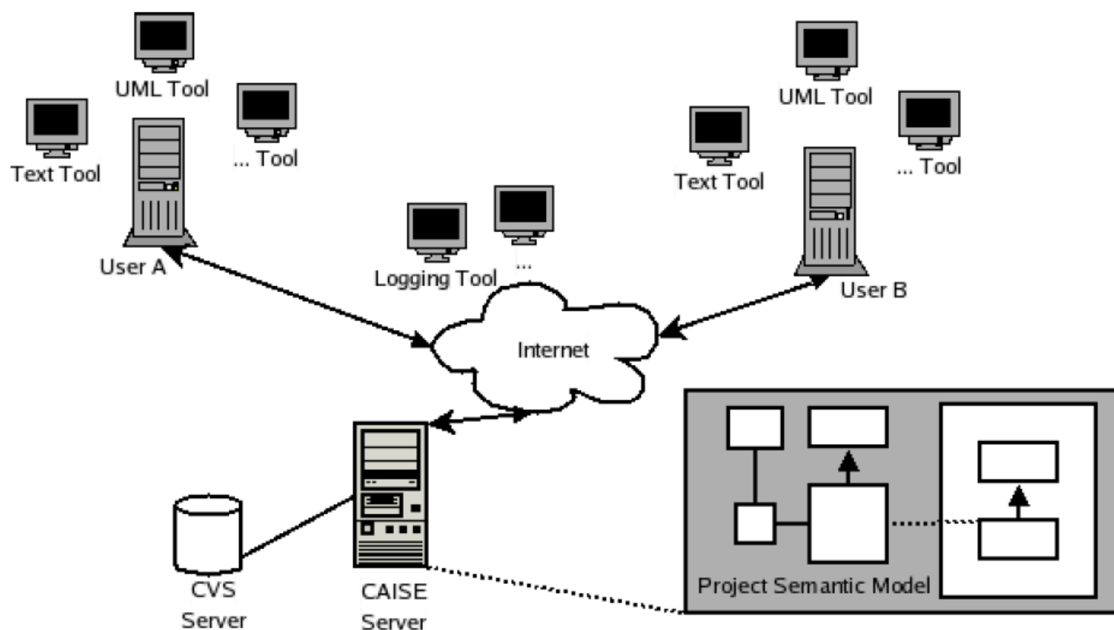


Рисунок 3.3 – Загальне схематичне зображення структури Caise

Щоб усунути труднощі створення корисних і зручних інструментів CSE, платформа Caise була розроблена для підтримки різних типів інструментів CSE. Враховуючи структуру, яка надає загальні служби співпраці через центральний сервер, передумова полягає в тому, що має бути можливість швидко створювати інструменти багатьох різних типів.

Даний підхід має перевагу, оскільки він дозволяє зробити основну архітектуру Caise відносно простою, бо надмірна архітектура може бути непрацездатною та представляти занадто багато часу для навчання для розробників практичних інструментів CSE. За замовчуванням структура Caise

робить не багато: підтримує загальний обмін артефактами, базову модель подій, міжпроцесний зв’язок і засоби для включення визначених користувачем операцій. Розробники зобов’язані надавати спеціальні інструменти, мовну підтримку та процедури аналізу за допомогою плагінів. Розширення, які можна включити, також включають зовнішні компоненти, такі як бібліотеки документів і системи відстеження помилок.

Існує можливість підтримувати низку нових спільних служб через структуру Caise. Це включає редагування артефактів у реальному часі, спільне семантичне моделювання проекту програмного забезпечення, детальне блокування семантичної моделі, а не керування версіями на основі файлів, запис повної діяльності розробки проекту з подальшою візуалізацією, інформацію про обізнаність користувачів у реальному часі і зворотній зв'язок, підтримка кількох мов, кілька переглядів артефактів, відсутність обмежень щодо типів програм, які можуть працювати разом над одним проектом, допуск для будь-якої кількості користувачів-учасників і динамічний збір показників.

Переваги впровадження інструментів CSE, що використовують підтримку на основі фреймворку є досить очевидними. Маючи центральний сервер, який контролює діяльність окремих інструментів, запити на модифікацію артефактів можна серіалізувати в стабільному порядку, що робить можливим і простим підтримувати редагування в реальному часі, включаючи засоби для складної проблеми спільного скасування [40]. Крім того, оскільки більшість функціональних можливостей реалізовано на сервері, інструменти клієнта відносно прості та легкі.

Огляд Caise Framework. Загальна концепція фреймворку Caise представлена на рисунку 3.4. Інструменти можуть приєднатися до проекту на основі Caise та розпочати редагування артефактів за допомогою протоколу інструментів Caise. API доступний для доступу до основної семантичної моделі та історії змін проекту. Інструменти CSE можна розробити швидко,

якщо дотримуватись протоколу інструментів Caise та використовувати семантичну модель як авторитетне джерело всієї інформації про проект.

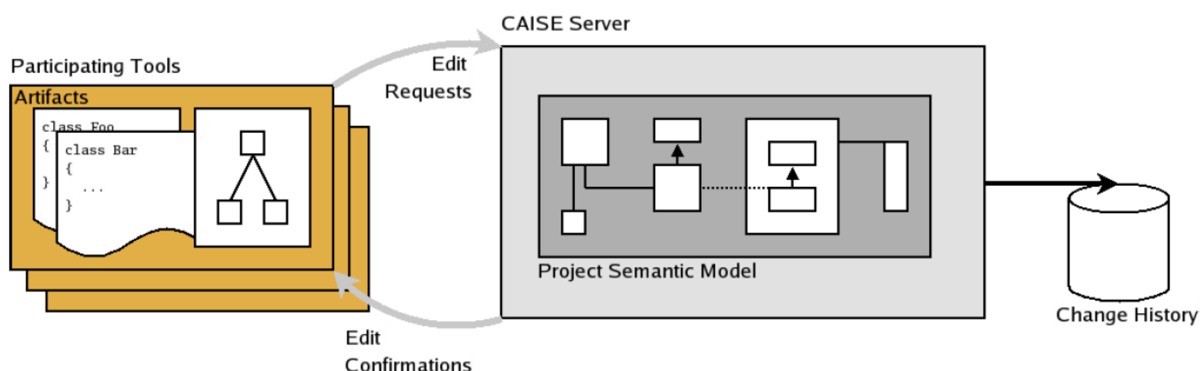


Рисунок 3.4 – Модифікація артефакту в рамках Caise

На рисунку 3.4. внутрішньою структурою є постійно оновлювана семантична модель, яка представляє основну структуру проекту програмного забезпечення та використовується для надання точної детальної інформації зворотного зв'язку інструментам, що беруть участь.

Логіка дизайну Caise та пов'язаних з ним інструментів полягає в тому, що перевага віддається безперервному спілкуванню та вирішенню конфліктів, а не приватній роботі з відстроченим виявленням проблем кодування. У рамках Caise приватна розробка на окремій копії вихідних файлів з подальшим а подальший процес злиття не підтримується явно. Навпаки, усі зміни в артефактах проекту поширюються на всю команду розробників програмного забезпечення, коли вони відбуваються.

Ключовим аспектом структури Caise є можливість генерувати детальну та точну інформацію, пов'язану з діяльністю користувачів, впливом змін і стосунками між користувачами. Оскільки всі дії кожного користувача спостерігаються центральним сервером, а всі модифікації артефактів семантично аналізуються, можна вивчити історію розробки програмного проекту до найтоншого рівня деталей.

Можливість обмінюватися артефактами в режимі реального часу, запроваджувати нові типи артефактів, відображати альтернативні види артефактів і запроваджувати нові типи зворотного зв'язку дозволяє підтримувати типи інструментів, розглянуті в попередньому розділі. Крім того, враховуючи багату семантичну модель програмного забезпечення, що міститься в кожному проекті на основі Caise , можна генерувати будь-який тип інформації зворотного зв'язку.

Архітектура. Caise не є спеціальним інструментом або IDE. Caise надає послуги CSE та семантичну модель програмного забезпечення, що дозволяє інструментам співпрацювати в режимі реального часу. Розробники інструментів можуть використовувати Caise будь-яким способом, який вважають за потрібне, і за бажанням розширювати структуру, наприклад додавати нові типи інформації для відгуків для інструментів CSE.

Структура Caise — це реалізація системи, яка включає функціональні аспекти. Замість побудови спеціальних інструментів, які відповідають заданому переліку вимог, було створено структуру, яка надає основні послуги для інструментів CSE, дозволяючи використовувати будь-який тип інструментів CSE.

Під час дослідження інструментів CSE визначено, що для аналізу вихідного коду, який розвивається в реальному часі, потрібна велика кількість обробки. Це було основним фактором, який керував рішенням про реалізацію спільної структури з центральним сервером. Таким чином, сервер Caise є, по суті, спільним механізмом IDE, де кожен інструмент на основі Caise є клієнтом. Єдиними спеціальними вимогами є мережеве підключення з низькою затримкою, наприклад комутована локальна мережа Ethernet і відносно потужне обладнання для розміщення сервера Caise.

Caise — це велика система, розроблена відповідно до чітко визначеного набору вимог. Він розширюваний, настроюваний і дуже універсальний. Слід зазначити, що хоча структура Caise може підтримувати будь-яку кількість різних мов, окремі проекти зазвичай базуються на одній мові.

Дизайн, орієнтований на код. Структура Caise розроблена як система, орієнтована на код згідно семантичної моделі, представляє структуру проекту програмного забезпечення, яка зазвичай походить від вихідних файлів проекту програмного забезпечення. Хоча відповідні альтернативні види вихідного коду, такі як діаграми класів UML, також можуть базуватися безпосередньо на семантичній моделі програмного забезпечення, це не означає, що всі типи артефактів SE можуть підтримуватися нативно. Наприклад, інструмент побудови діаграм UML може отримати більшу частину інформації, яка потрібна для діаграми компонентів [38] із семантичної моделі, такої як назви класів і пакетів, а також асоціації між пакетами. Однак поняття діаграми компонентів вищого рівня, такі як ключові компоненти та зв’язки системи, не можуть бути автоматично виведені, оскільки вони явно чи неявно не представлені в семантичній моделі.

Підхід Caise, орієнтований на код, добре підходить для структури, яка підтримує інструменти розробки CSE; багато популярних інструментів, що використовуються сьогодні, є кодоцентричними, наприклад Eclipse і Visual Studio і часто використовуються в режимах лише коду. Передбачається, що більшість інструментів CSE знаходяться на стадіях впровадження, тестування та обслуговування життєвого циклу SE і згодом вони будуть побудовані на прямій маніпуляції з вихідним кодом і діаграмами на основі семантичної моделі. Підхід, орієнтований на код, дозволяє інструментам CSE взаємодіяти з іншими інструментами та базовою структурою без потреби в анотаціях коду чи складних інтерфейсах обміну повідомленнями. Крім того, семантичну модель можна використовувати як канонічне джерело інформації, забезпечуючи узгодженість між інструментами.

Маючи структуру, орієнтовану на код, інструменти, орієнтовані на код, такі як текстові редактори, налагоджувачі та інструменти діаграм класів, є найлегшими для підтримки. Такі інструменти, як діаграми стану та взаємодії/послідовності, також добре підтримуються, але вимагатимуть деякої додаткової інформації, такої як дані макета, які надаються зовнішніми

засобами. Найскладніші типи інструментів SE, які підтримуються в рамках Caise, пов’язані з робочим процесом, наприклад діаграми варіантів використання, оскільки такі концепції, як потоки процесів, актори та клієнти, ніколи не моделюються в рамках основної структури програмного забезпечення. У цих випадках структуру Caise можна розширити шляхом введення нових типів інструментальних артефактів, шляхом розширення семантичної моделі за межі рівня вихідного коду або навіть за допомогою використання іншої семантичної моделі, розробленої спеціально для цього класу інструментів.

Методології програмної інженерії. Інфраструктура Caise не накладає конкретну методологію на інструменти на основі Caise, а розробники інструментів можуть запроваджувати певні методології поверх інструментів CSE, якщо це буде потрібно. Ключовим проектним рішенням було уникати нав’язування будь-якої конкретної парадигми програмування, тому ключовою метою фреймворку Caise є підтримка загальної спільної розробки програмного забезпечення. Такі процеси, як RUP або XP, потенційно можуть застосовуватися на сервері Caise і розробники інструментів на основі Caise можуть вільно впроваджувати будь-які специфічні для процесу механізми в межах свого набору інструментів.

Caise ідеально підходить для підтримки розподіленого парного програмування. За допомогою Caise цю практику можна назвати N-програмуванням, оскільки немає теоретичних обмежень щодо кількості людей і типів інструментів, які можуть співпрацювати в будь-який момент часу. Це значна перевага перед звичайним парним програмуванням.

Ступінь співпраці. Важко забезпечити повністю синхронну службу для редагування вихідних файлів та інших артефактів SE. Кілька інструментів, які підтримують спільне редагування, такі як JBuilder [12] працюють із дуже обмеженими політиками контролю рівня, такими як передача маркерів. Метою структури Caise є підтримка будь-якої кількості користувачів, які співпрацюють у режимі реального часу.

Щоб проілюструвати ступінь співпраці, яку пропонує Caise на рисунку 3.5 представлено інструменти CSE відносно спектру співпраці.

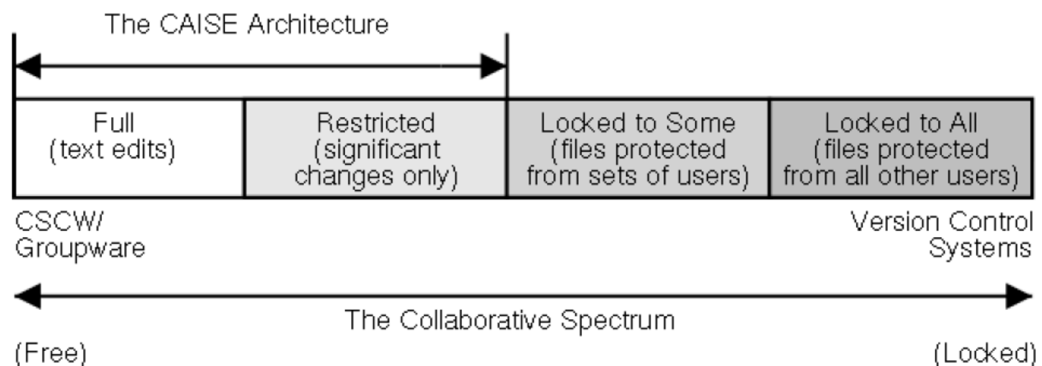


Рисунок 3.5 – Структура Caise в контексті спільного використання

Як показано на цьому рисунку, інструменти на основі Caise мають певні варіації в обсязі співпраці, яку вони підтримують. Наприклад, більшість інструментів CSE підтримуватимуть повне спільне редагування артефактів, але деякі інструменти можуть вирішувати поширювати лише важливі події, такі як завершені тіла методів.

Як також показано на рисунку 3.5, інструменти на основі Caise не розроблені для підтримки звичайних режимів SE, таких як оптимістичне або песимістичне блокування файлів. Звичайний SE базується на ідіомі копіювання, модифікації та злиття систем сховищ коду, але завдяки наявності повністю синхронного редагування артефактів інструменти Caise зазвичай працюють на центральних спільних артефактах із соціальними протоколами для полегшення посередництва між розробниками.

3.3 Архітектурне проектування та розробка семантичної моделі фреймоворку командної розробки

Огляд архітектури структури Caise , включно з інструментами CSE, які беруть участь, представлено на рисунку 3.6.

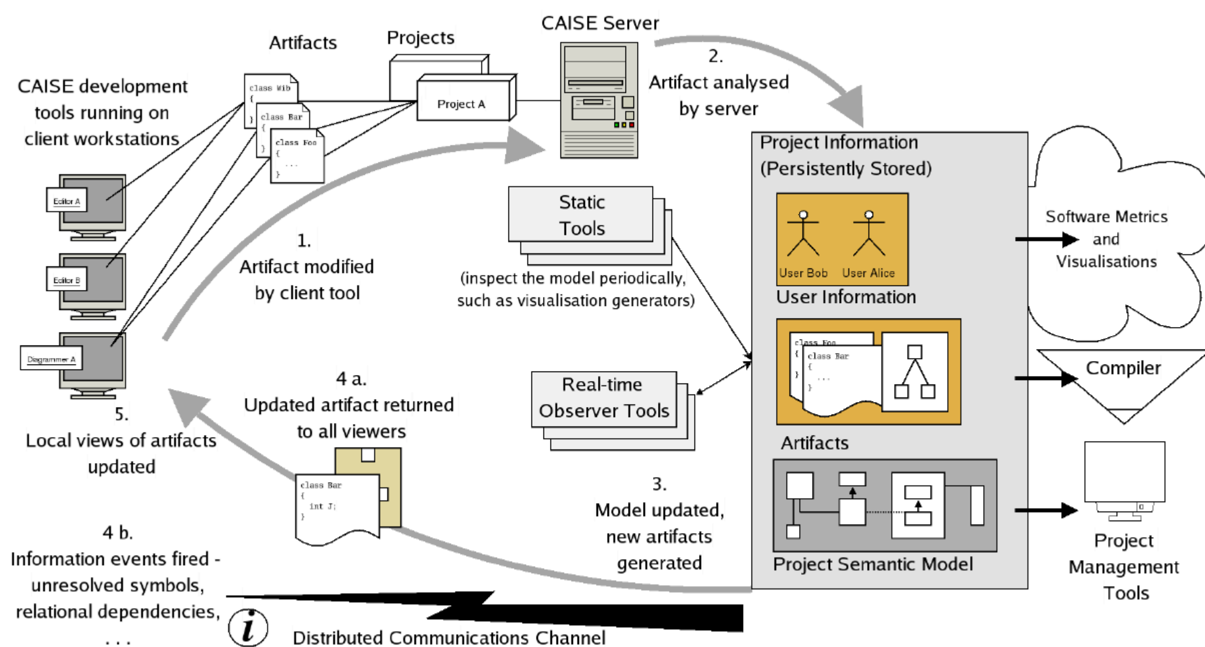


Рисунок 3.6 – Представлення фреймворку Caise та основних складових інструментів

На цьому рисунку показано інструменти на основі Caise, які оновлюють артефакти (1), які в свою чергу доставляються на сервер Caise. Сервер аналізує артефакти (2) і оновлює базову семантичну модель (3). Оновлені артефакти повертаються до кожного інструменту CSE (4a), а динамічний зворотний зв’язок, наприклад інформація про близькість користувача, також повертається на основі подій (4b). Після отримання оновлених артефактів інструменти коригують свої локальні погляди на проект (5).

Щоб пояснити взаємозв’язки між артефактами, інструментами, семантичною моделлю та плагінами зворотного зв’язку, рисунок 3.7 ілюструє ключові особливості між компонентами в рамках Caise.

Користувачі в рамках проекту можуть працювати з різними типами інструментів одночасно. Інструмент зазвичай працює з одним типом артефакту за раз, хоча може бути розміщено кілька артефактів одного типу. IDE може включати кілька інструментів в одній програмі, але це функція, яка не залежить від структури Caise . Плагіни зворотного зв’язку є специфічними для семантичної моделі та можуть створювати загальну інформацію

зворотного зв’язку або залежно від інструменту залежно від реалізації. Кожен проект має один екземпляр семантичної моделі, але сервер Caise може підтримувати більше одного типу семантичної моделі.

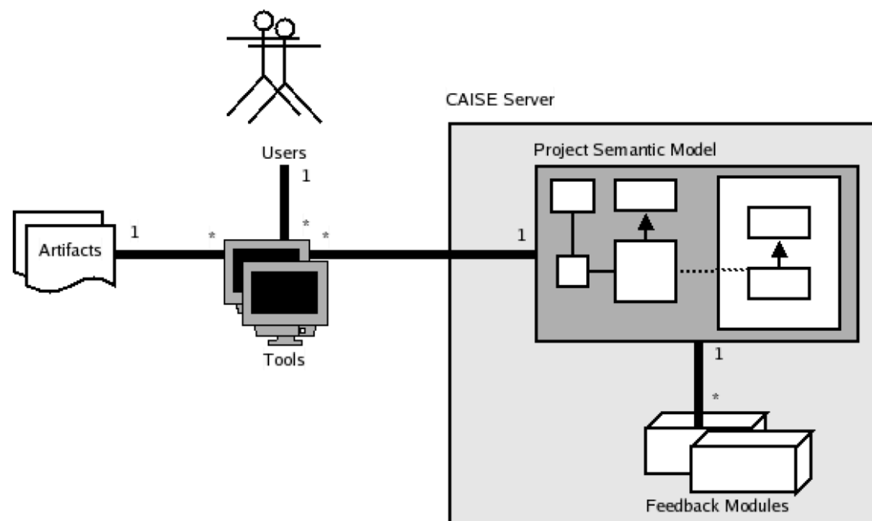


Рисунок 3.7 – Зв’язки між ключовими компонентами структури Caise

Сама структура Caise не робить жодних припущень щодо типів інструментів, семантичних моделей і модулів зворотного зв’язку. Його роль полягає в координації компонентів фреймворку на основі добре відомих інтерфейсів, у підтримці зв’язку між інструментами, у сприянні розповсюдженню подій між інструментами та сервером, а також у забезпеченні зберігання та спільного доступу до артефактів на основі Caise . Інструмент підтримки аналізаторів і плагінів зворотного зв’язку, а також операції, що залежать від мови.

Семантична модель проекту. Переваги семантичного моделювання включають можливість швидкого рефакторингу одиниць коду, точного запиту семантичної моделі в реальному часі та визначення зв’язків між користувачами та одиницями коду. Семантична модель для кожного проекту на основі Caise зберігається на сервері Caise. Концепція використання спільної семантичної моделі для полегшення співпраці між інструментами SE є новою для галузі CSE.

Семантична модель представляє всі сутності програмного забезпечення, такі як пакети, класи та методи, а також зв’язки між ними, такі як виклики методів і залежності коду.

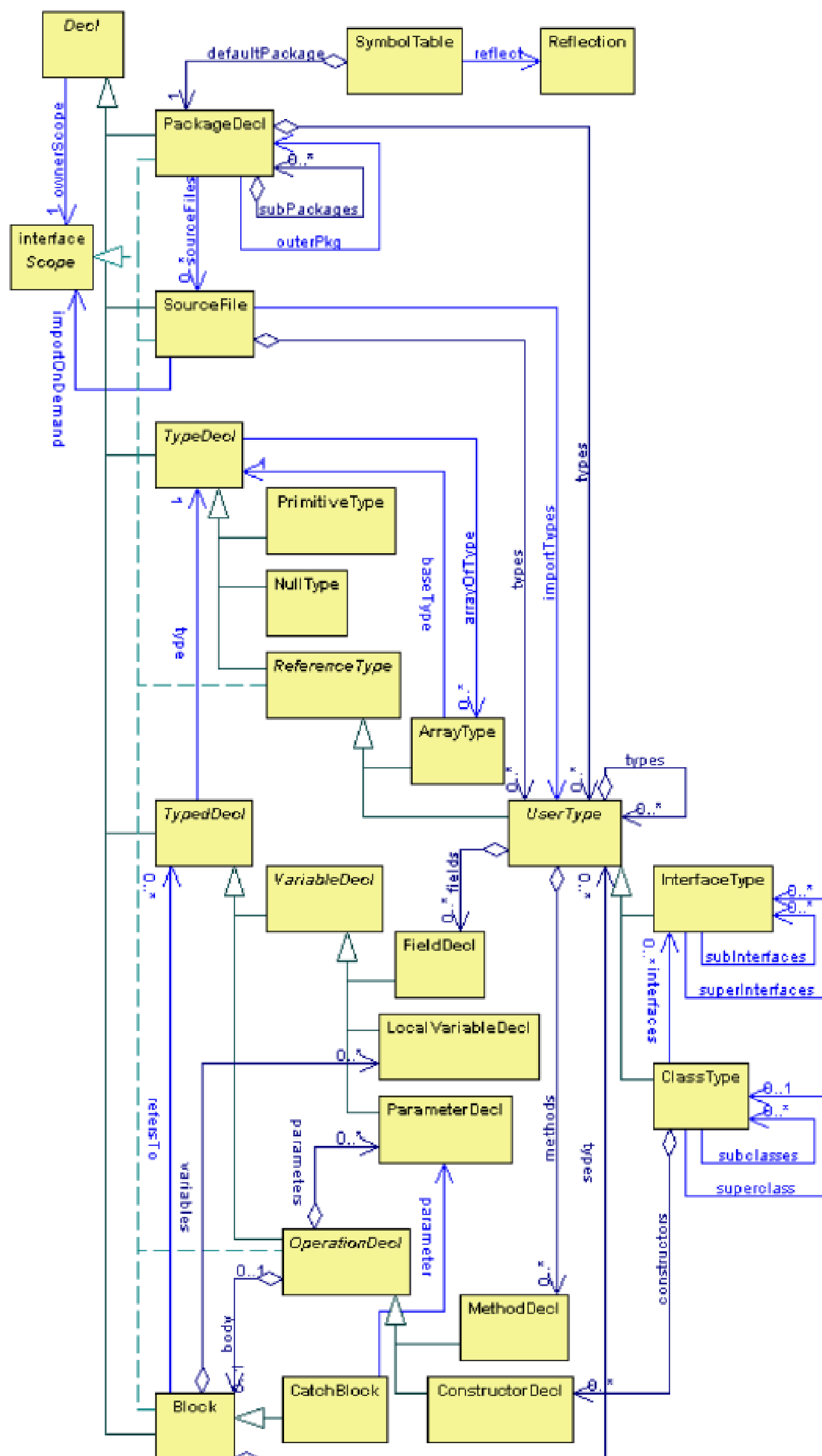


Рисунок 3.8 – Семантична модель ОО забезпечення в нотації UML

Ключовою особливістю дизайну сервера Caise є розмежування мов і семантичної моделі програмного забезпечення. Хоча вихідні файли та інструменти на основі Caise можуть мати певну мову, семантична модель не залежить від мови. Це означає, що інструменти, які перевіряють семантичну модель, такі як модулі зворотного зв'язку, можуть бути написані незалежно від конкретних мов, що збільшує кількість їх використання в рамках Caise.

Оскільки дослідження базуються в основному на підтримці Java та Java-подібних мов, основна семантична модель, яка зараз використовується в Caise, базується на ООП. Ця семантична модель схожа на структуру .Net [20], де також можна охопити кілька мов.

Для мов, які принципово відрізняються від парадигми ООП, інший тип семантичної моделі може бути введений у структуру Caise або існуюча семантична модель може бути розширена.

Поточна семантична модель програмного забезпечення, запропонована Caise, забезпечує повну підтримку Java. Архітектурна діаграма для семантичної моделі Java представлена на рисунку 3.8.

Точна семантична модель програмного забезпечення була необхідна для побудови фреймворку Caise і замість написання окремо, була сформована нова версія семантичної моделі JST. Семантична модель JST була використана, оскільки вона точно моделює програми Java, базуючись безпосередньо на стандартній граматиці експозиції Java [47] і має логічний API для навігації семантичною моделлю.

Розширення, необхідні для використання в рамках Caise, включали підтримку поступових оновлень семантичної моделі, розширення явно зіставлених зв'язків у семантичній моделі та обчислення списку семантичних змін на основі оновлень семантичної моделі.

Семантична модель програмного забезпечення в рамках Caise розроблена для перевірки поступового оновлення та запитів користувачів. Це відрізняється від семантичних моделей у Borland's Together Architect та Eclipse IDE, оскільки ці семантичні моделі забезпечують обмежений

програмний доступ і мають мало документації. Крім того, семантична модель у Caise підтримує пряму модифікацію всієї семантичної моделі.

Приклади процедур, доступних через API для семантичної моделі програмного забезпечення Caise , включають: lookupType() , get/addPackage() , get/addDeclaration() , get/addSourceFiles() , get/addType() і get/addMethod() . Ці підпрограми такі ж точні, як і будь-який інший компілятор і їх можна викликати з будь-якого інструменту на основі Caise.

Семантична модель програмного забезпечення Caise моделює вихідний код Java до рівня оператора. Єдиними низькорівневими конструкціями, які явно не моделюються, є оператори керування, такі як оператори if , цикли for та оператори switch. Цей вибір дизайну було зроблено автором оригінальної семантичної моделі, і неможливість моделювання цих низькорівневих концепцій не впливає на структуру Caise будь-яким значним чином. Оголошення та використання всіх типів все ще моделюються, навіть до рівня локальних змінних; наприклад, оператор if безпосередньо не моделюється, але будь-які використання змінних, включаючи оголошення та призначення в операторі. Поточна семантична модель у Caise може бути використана розробниками інструментів для більшості поширених мов. Однак семантичну модель можна розширити, щоб охопити інші концепції, наприклад низькорівневі керуючі оператори або абстрактні компоненти, такі як актори UML і переходи станів.

Якщо для проекту, заснованого на Caise, потрібен інший тип семантичної моделі, швидше за все, через нетрадиційну мову або використання Caise для зовсім іншого домену, наприклад розробки веб-сайтів, будь-які існуючі інструменти та плагіни зворотного зв’язку, необхідні для використання необхідно оновити відповідно до властивостей і структури нової семантичної моделі.

Семантичні аналізатори в рамках Caise відповідають за побудову та підтримку семантичної моделі для кожного програмного проекту. Основним завданням Caise - сумісних аналізаторів є перевірка дерев синтаксичного

аналізу та вставка будь-яких ідентифікованих декларацій у семантичну модель проекту.

Аналізатору потрібно виконати багато роботи, щоб побудувати семантичну модель програмного забезпечення. Однак більшу частину цієї роботи можна виконати незалежно від мови. Таким чином, мовні аналізатори не відповідають виключно за побудову семантичної моделі програмного забезпечення. Загальна семантична модель Caise містить не лише оголошення та зв'язки в проекті програмного забезпечення, але й підпрограми для пошуку типів і побудови більшої частини самої семантичної моделі.

Багаторівнева архітектура. Структуру Caise можна точно описати в термінах трьох рівнів, як показано на рисунку 3.9. Ці рівні: співпраця в межах кожного типу інструменту, співпраця між кожним типом інструменту та основні функції SE. Ці три рівні також можна описати як CSCW, CSE та SE.

Фреймворк Caise являє собою трирівневу архітектуру для зменшення складності сервера Caise. Якби сервер Caise повністю відповідав за взаємодію між усіма типами та екземплярами інструментів у проекті, розробка та реалізація сервера були б дуже складними. Крім того, інструменти на основі Caise також було б дуже важко розробити, оскільки вони не були б відокремлені від функцій інших типів інструментів.

На рівні CSCW структури Caise, як показано на рисунку 3.9, синтаксичні події, специфічні для кожного типу інструменту, містяться в межах цього інструменту. Іншими словами, цей рівень відповідає за підтримку повної синхронізації всіх екземплярів кожного типу інструментів. Наприклад, кожно-символьні події, створені в результаті модифікації вихідного файлу, поширюються на всі інші текстові редактори в рамках, але не на інші інструменти безпосередньо. Подібним чином, якщо клас у інструменті створення діаграми переміщується в нове місце в межах діаграми, сповіщаються лише інструменти, які спільно використовують цей конкретний вигляд діаграми. Сервер Caise внутрішньо знає про всі такі зміни,

але ці події низького рівня або події на основі CSCW поширюються лише на відповідні інструменти в рамках проекту.

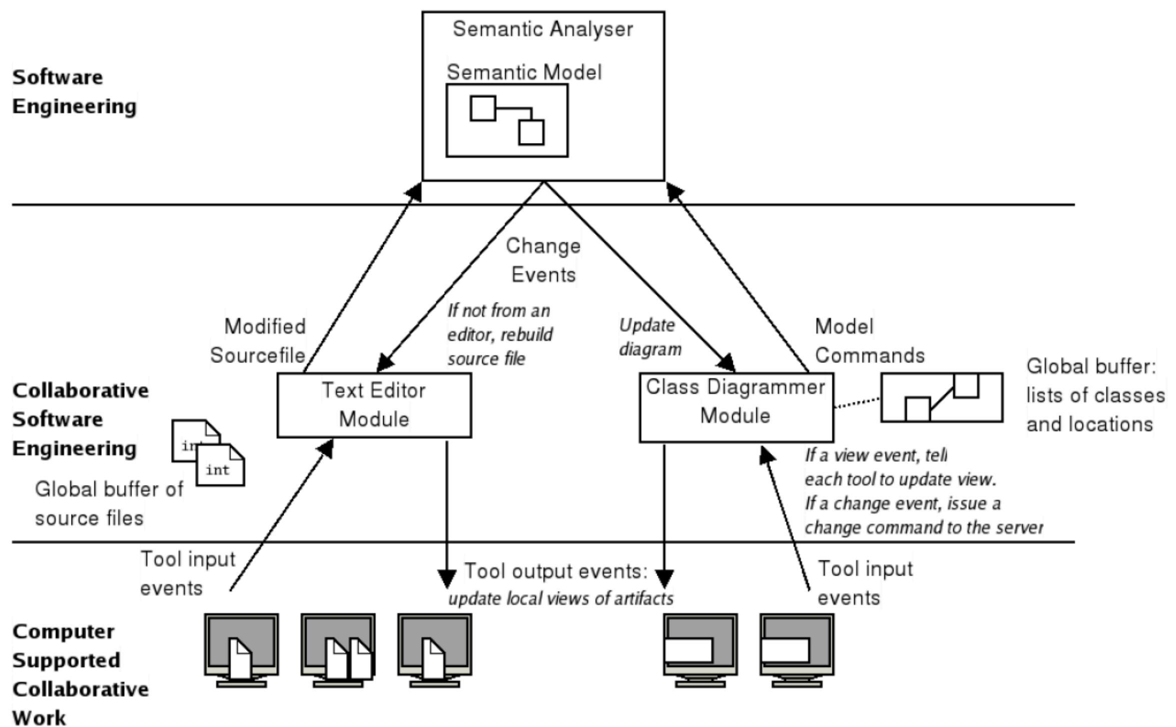


Рисунок 3.9 – Три концептуальні рівні структури Caise

На рівні CSE фреймворку Caise фреймворк розглядає події, пов’язані із семантичними змінами в проекті програмного забезпечення; це унікальна особливість підходу Caise. Наприклад, якщо інструмент редактора коду отримує натискання клавіш, які зрештою утворюють новий метод або оголошення класу, ця семантична подія поширюється на всі інші типи інструментів. Це дозволяє розробникам діаграм класів та іншим інструментам помічати семантичну зміну та відповідним чином оновлювати власні погляди на проект.

На рівні SE фреймворку Caise фреймворк підтримує програмний продукт, над яким працюють інструменти на основі Caise. Відокремлюючи функції CSE та CSCW на інших рівнях, рівень SE може зосередитися на аналізі коду, створенні виконуваних файлів і поступовій інтеграції оновлених вихідних файлів. У межах Caise загальна семантична модель програмного

забезпечення є достатньо виразною, щоб представити погляди на артефакти як вихідні файли або діаграми; це видно з демонстрацій інструментів, представлених на супровідному диску ресурсів.

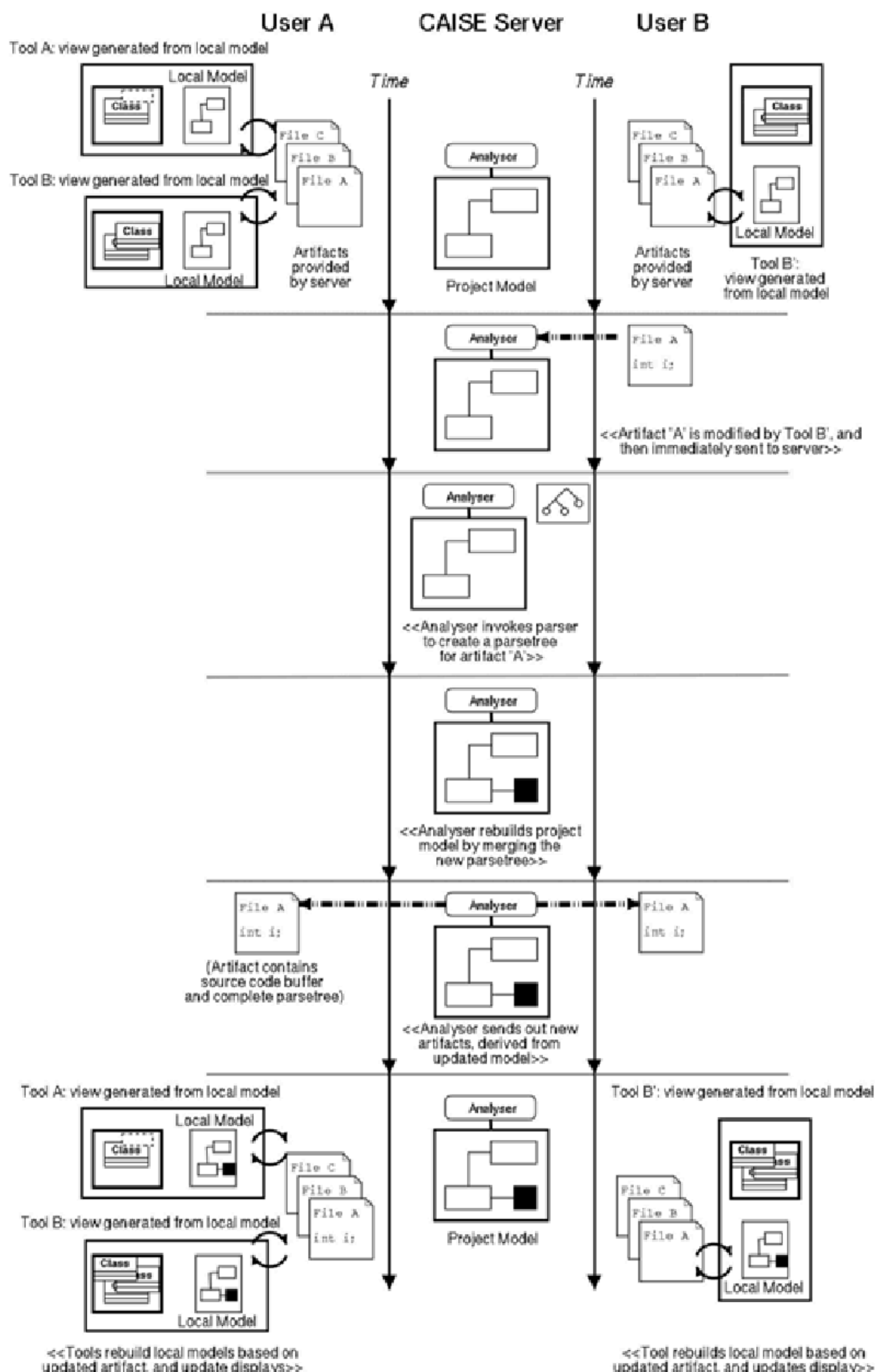


Рисунок 3.10 – Схематичний вигляд модифікації артефакту в Caise з сервером Caise

Як правило, після синтаксично правильних модифікацій оновлені дерева синтаксичного аналізу та буфери коду розподіляються між інструментами, що дозволяє їм оновлювати власні локальні перегляди артефактів, як показано на рисунку 3.10.

Наприклад, якщо інструмент діаграми класів додає новий метод до існуючого класу, отриманий оновлений вихідний файл, який містить цей клас, буде надіслано всім текстовим редакторам для оновлення їхніх власних представлень.

3.4 Подання моделі події та модифікації артефакту при використанні фрейворку командної розробки

Фреймворк Caise спирається на прості та часті події від інструментів CSE, які вони використовують. Інструменти на основі Caise повідомляють серверу про кожну виконану дію, наприклад зміну розташування курсору або натискання клавіші, і сервер відповідно оновлює базову модель. Це означає, що сервер має багато частих обробок, але кожне завдання є відносно невеликим. Працюючи з дуже детальними подіями, платформа Caise також може забезпечити синхронну співпрацю, наприклад редагування спільного тексту в реальному часі та негайний аналіз змін коду.

Окрім вхідних подій від інструментів, сервер Caise генерує вихідні події, які надсилаються інструментам та іншим зацікавленим програмам. Вихідні події включають сповіщення про останні модифікації артефактів і події зворотного зв’язку, такі як метрики, звіти про вплив і інформацію про сусідство коду.

Модель події для фреймворку Caise представлена на рисунку 3.11. У моделі подій Caise сервер Caise транслює події різних типів усім залученим інструментам, які зареєстровані як слухачі подій. Інструменти можуть реєструватися як слухачі для всіх подій або лише для окремих категорій подій. Події містять деталі загальної дії, як-от артефакт, редагований

вказаним користувачем, і конкретні деталі, наприклад ураженний текст і зміщення файлу. Кожна подія, створена в рамках проекту на основі Caise , також записується в журнал подій Caise.

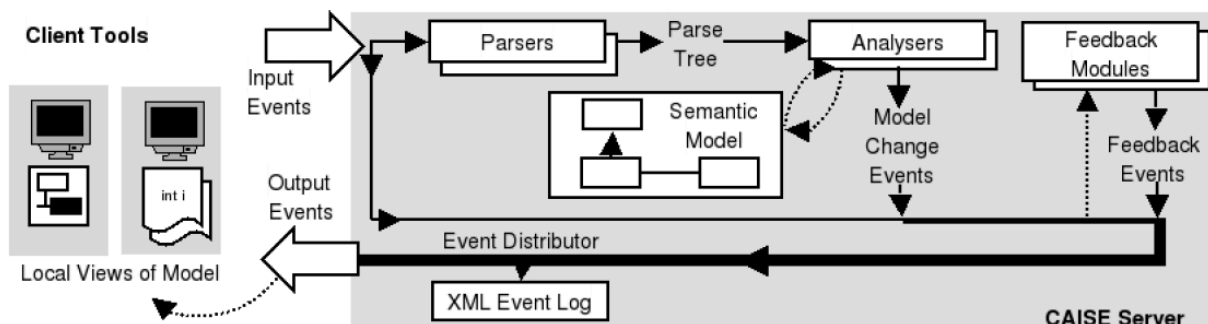


Рисунок 3.11 – Модель події Caise

Кожен тип події Caise коротко підсумовано в таблиці 3.1. Повнофункціональні інструменти реєструються як слухачі для всіх подій, а інші компоненти, як-от графік змін зацікавлені лише в конкретних типах подій.

Таблиця 3.1

Типи подій у структурі Caise

Type	Typical actions
Project	A project is created or deleted.
Artifact	An artifact is added, removed or edited.
Chat	A user issues text or audio messages.
Feedback	Tool-specific custom units of information exchange.
Client	A client opens, closes or moves location within an artifact, or rebuilds a project.
Change	The project's semantic model is manipulated directly or via artifact modification.

Основна структура подій Caise така. Кожна подія записує користувача, відповідального за створення події, користувачів, які отримали подію, тип

події, час її створення, посилання на будь-яку семантичну компоненту моделі, безпосередньо пов’язані з подією, будь-які інші дані, пов’язані з подією.

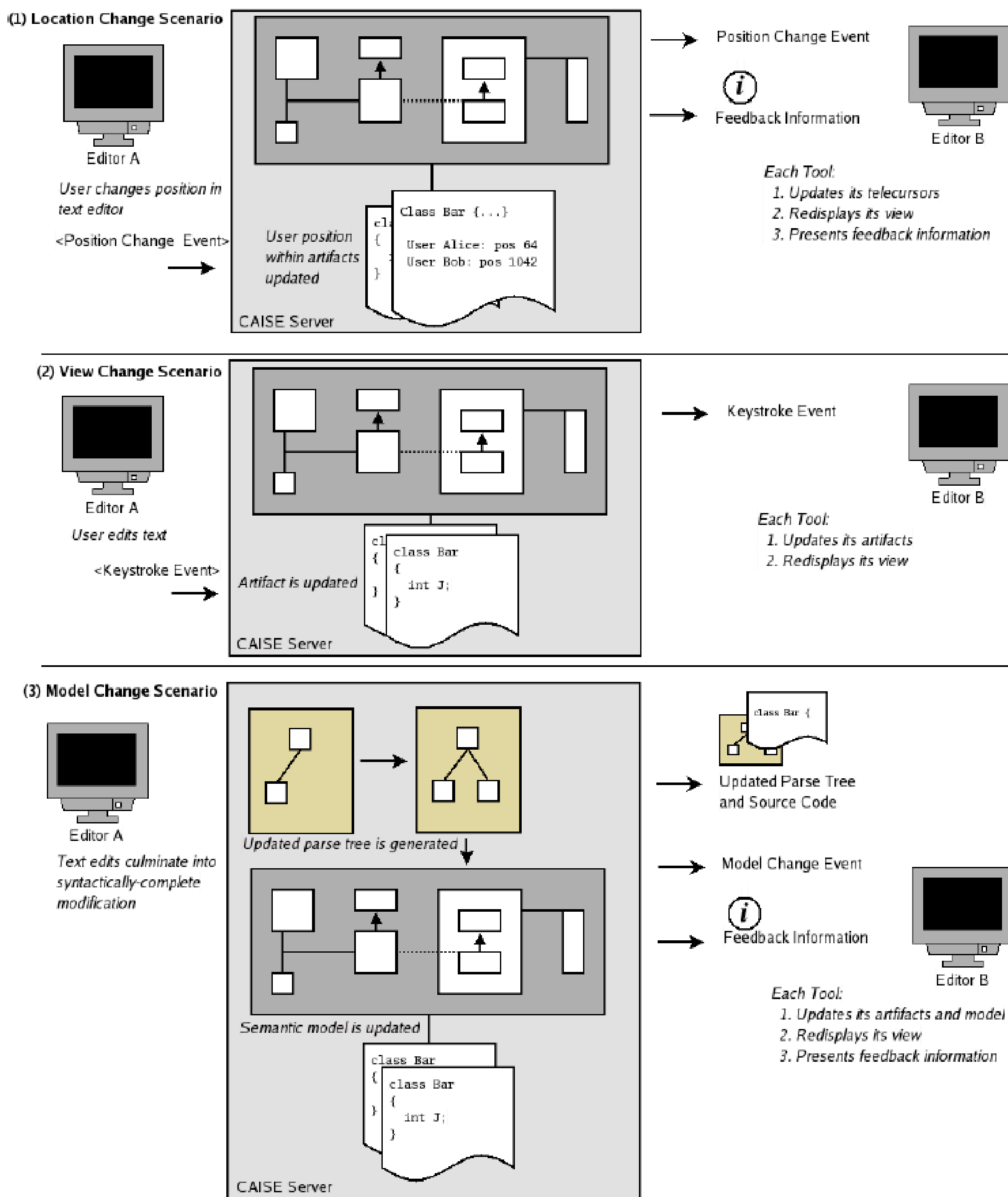


Рисунок 3.12 – Ключові типи дій у рамках Caise протоколу (при синхронному інструменті, такому як текстовий редактор, редагування буде атомарною подією — одним натисканням клавіші.)

З точки зору спільного використання файлів, структура Caise базується на шаблоні атомарної інтеграції, як представлено в другому розділі. Кожен інструмент на основі Caise працює зі спільним набором артефактів, які зберігаються на центральному сервері Caise. Артефакти обмінюються та змінюються в режимі реального часу, що означає, що зміни відтворюються на всіх інструментах у системі, коли вони відбуваються.

На рисунку 3.12 представлено ключові типи модифікацій артефактів. До них належать зміни в розташуваннях користувачів (1), зміни в синтаксисі артефакту (2) і семантичні зміни, які є результатом синтаксичних змін (3). Цей малюнок також ілюструє заснований на подіях характер фреймворку Caise — інструмент генерує вхідну подію, сервер Caise відповідає, оновлюючи відповідні артефакти, і, нарешті, інструменти на основі Caise оновлюють свої локальні представлення на основі відгуків від сервера Caise.

Коли інструмент на основі Caise визначає, що відбулася модифікація артефакту, наприклад вихідний файл піддається редагуванню, інструмент сповістить сервер Caise про модифікацію.

Коли інструмент визначає, що файл може бути в синтаксично правильному стані, сервер Caise викликається для аналізу файлу. Після успішного синтаксичного аналізу зміни семантично аналізуються, а дерева синтаксичного аналізу розповсюджуються на всі інструменти-учасники, яким вони потрібні, що дозволяє їм оновлювати власні локальні уявлення про проект, як показано на рисунку 3.10.

Якщо семантичну модель проекту було змінено в результаті модифікації, ця інформація також збирається для розповсюдження як події зворотного зв’язку для будь-яких зацікавлених інструментів. Якщо файл не вдається проаналізувати, це зазначається в журналі подій проекту Caise , і інформаційна подія транслюється для будь-якого інструменту, який може збирати показники діяльності проекту.

У більшості випадків лише синтаксично правильні модифікації поширюються на інші типи інструментів. Наприклад, коли нова властивість

вводиться з редактора коду, ця часткова та синтаксично неправильна декларація не передається іншим інструментам, таким як діаграми класів. Цей підхід викликає транзакційну проблему: якщо один користувач наразі вводить рядок коду за допомогою інструмента редагування тексту, а другий користувач одночасно вносить зміни до відповідної сутності в інструменті діаграми, Caise сервер наразі не інтегрує незафіксовані зміни текстового редактора з оновленим деревом аналізу, що призводить до втрати будь-яких незафіксованих змін текстового редактора.

У рідкісних ситуаціях, коли протоколи не забезпечують належний захист від суперечливих модифікацій між спільними артефактами, проблему втрати тексту модифікації можна легко вирішити. Це досягається додаванням механізму на сервері Caise, який просто об'єднує будь-які незафіксовані зміни в існуючому вихідному файлі в новосформоване дерево аналізу та оновлений буфер вихідного коду. Крім того, окремі засоби редагування тексту можуть легко реалізувати механізм об'єднання, який виконує той самий процес на стороні клієнта.

Розробники інструментів також можуть ігнорувати зміни в артефактах, внесені іншими користувачами, щоб забезпечити певний ступінь ізоляції для програміста, але це ризикує мати вихідні файли, які більше не синхронізуються з сервером.

Для підтримки розробки інструментів CSE базова структура повинна забезпечувати основні функції SE, а також засоби міжпроцесного зв'язку та компоненти групового програмного забезпечення. У рамках Caise всі ці можливості підтримуються сервером Caise .

Сервер Caise відповідає за зберігання та спільне керування всіма артефактами в рамках проекту на основі Caise. Сервер Caise також керує семантичною моделлю програмного забезпечення, журналом подій проекту та інформацією про користувача. Інші функції сервера Caise включають передачу різних типів подій відповідним слухачам, таким як інструменти розробки, інструменти управління проектами та генератори візуалізації.

Сервер Caise також визначає механізми для підтримки розширюваності, такі як введення нових мов, артефактів і типів зворотного зв'язку з інфраструктурою.

Щоб підтримувати справді корисні інструменти CSE на основі Caise, основна структура має бути високої якості. Це означає, що сервер Caise повинен мати можливість безперебійно обробляти кілька одночасних запитів у будь-який час, забезпечувати підтримку скасування в рамках спільної роботи, мати прийнятний час відповіді навіть за великого навантаження на систему та бути практичним для розширення.

Надсилання запитів на модифікацію на сервер Caise нечасто та періодично як пакет не рекомендується в рамках Caise , оскільки це порушує базовий протокол інструменту Caise. Сервер Caise можна розширити, щоб підтримувати це, використовуючи звичайні засоби контролю транзакцій, але це не входить до сфери дії поточної версії Caise та не узгоджується з принципом CSE у реальному часі.

Фреймворк Caise базується на централізованій серверній архітектурі. Цей вибір дизайну був зроблений через необхідність; на початку цього дослідницького проекту апаратне забезпечення настільного комп'ютера було недостатньо потужним, щоб виконувати обробку подій оновлення інструментів, аналіз, семантичний аналіз і оновлення семантичної моделі проекту.

Сьогодні апаратне забезпечення високого класу здатне запускати серверний процес Caise. Таким чином, за потреби можна підтримувати розподілену версію архітектури Caise , яка добре підійде для проектів з відкритим вихідним кодом і багатонаціональних команд розробників, оскільки вони стають все більш розподіленими за своєю природою. Найсуттєвішою зміною поточної архітектури було б застосування розподіленого алгоритму керування паралелізмом для підтримки синхронізації між інструментами, щоб замінити центральний механізм черги, який зараз використовується на сервері Caise.

Висновки до розділу 3

Отже, в цьому розділі запропоновано ряд аспектів дизайну, які слід враховувати під час створення інструментів CSE на основі шаблонів. Запропонований підхід до проектування, прийнятий для розміщення великої кількості розробників і баз коду, вимагає ретельного балансування з рівним урахуванням суперечливих факторів, таких як обізнаність інших проти безперебійних режимів роботи. Запропоновано фреймоврк спільної розробки CAISE, що забезпечує хороший рівень співпраці в рамках розробки програмного забезпечення. CAISE пропонує новий підхід до підтримки розробки шляхом семантичного моделювання, адаптації нових мов та інструментів, підтримки масштабованості та можливості налаштування та розширення. Розроблена семантична модель CAISE, журнал подій і артефакти які є джерелами інформації для аналізу, а структура забезпечує середовище реального часу, побудоване на подіях, для управління спільними програмними проектами. Представлено архітектурний дизайн CAISE, включаючи ключові характеристики фреймворку та принципи проектування.

ВИСНОВКИ

В кваліфікаційній роботі досліджено процес підвищення ефективності застосування семантичної моделі групової динаміки команд розробників програмного забезпечення та сервісів. Запропоновані моделі співпраці які були визначені як корисний засіб для опису тенденцій взаємодії та співпраці. Представлено опис шаблонів для підтримки CSE, загальні моделі взаємодії, типи співпраці в рамках розробки і досліджено шаблони-кандидатів для CSE.

Дослідження щодо шаблонів CSE, узгоджується зі способом визначення шаблонів у суміжних предметних галузей. Представлена тут класифікація, не є єдиним способом групування шаблонів, пов’язаних із CSE, але вона відразу допомагає в описі вимог до майбутніх інструментів CSE.

Одним із найважливіших аспектів шаблонів CSE є те, що вони надають розробникам інструментів цінну інформацію про дизайн, яку інакше було б важко отримати. Знання про шаблони, пов’язані з CSE, дозволяє дослідникам зосередитися на правильному розробці фундаментального дизайну інструментів CSE та допоміжних засобів.

Визначені закономірності, такі як Mode of Development, які було важко ідентифікувати, оскільки професійні інструменти CSE ще не мають широкого використання. Однак цілком ймовірно, що ці способи розробки підтримуватимуться інструментами CSE, коли з’являться нові інструменти. Швидкість засвоєння цих шаблонів буде найточнішим визначальним фактором успішної ідентифікації шаблонів.

В роботі запропоновано підхід спільного семантичного моделювання для підтримки CSE, який показано на структурі CAISE , де різні типи потужних і раніше недоступних інструментів CSE можуть співпрацювати в режимі реального часу на спільному наборі артефактів. Було детально продемонстровано побудову та роботу кількох інструментів на основі CAlSE.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles Techniques and Tools. Addison Wesley, Reading, MA, 1988. ISBN 0-201-10194-7.

2. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. FiksdahlKing, and S. Angel. A Pattern Language: Towns, Buildings, Construction. Oxford University Press, 1977.

3. K. Baker, S. Greenberg, and C. Gutwin. Heuristic Evaluation of Groupware Based on the Mechanics of Collaboration. In M.R. Little and L. Nigay, editors, Proceedings of Engineering for Human-Computer Interaction, volume 2254 of Lecture Notes in Computer Science, pages 123–139, Toronto, Canada, May 2001. Springer-Verlag.

4. Kevin Baker, Saul Greenberg, and Carl Gutwin. Empirical Development of a Heuristic Evaluation Methodology for Shared Workspace Groupware. In Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work, pages 96–105. ACM Press, 2002. ISBN 1-58113-560-2.

5. Sergio Bandinelli, Elisabetta Di Nitto, and Alfonso Fuggetta. Supporting Cooperation in the SPADE-1 Environment. IEEE Transactions on Software Engineering, 22(12):841–865, 1996. ISSN 0098-5589.

6. Kent Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley, Reading, MA, 1st edition, October 1999.

7. James Begole, John C. Tang, and Rosco Hill. Rhythm Modeling, Visualizations and Applications. In Proceedings of the 16th annual ACM symposium on User interface software and technology, pages 11–20. ACM Press, 2003. ISBN 1-58113-636-6.

8. Brian Berliner. CVS II: Parallelizing Software Development. In Proceedings of the USENIX Winter 1990 Technical Conference, pages 341– 352, Berkeley, CA, 1990.

9. BitKeeper User Documentation. BitMover Incorporated, September 2005.

10. Marko Boger, Thorsten Sturm, Erich Schildhauer, and Elizabeth Graham. Poseidon for UML User Guide. Gentleware AG, 2002.

11. What's New In Borland JBuilder 2005. Borland Software Corporation, September 2004.

12. Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. An overview of PCTE and PCTE+. In SDE 3: Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, pages 248–257, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-290-X.

13. Lionel C. Briand, Christian Bunse, and John W. Daly. A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs. IEEE Transactions on Software Engineering, 27(6):513–530, 2001. ISSN 0098-5589.

14. Felix C. Brodbeck. Communication and Performance in Software Development Projects. European Journal of Work and Organizational Psychology, 10(1):73–94, March 2001.

15. Frederick P Brooks Jr. The Mythical Man Month: Essays on Software Engineering. Addison-Wesley, 2nd edition, 1995. ISBN 0-201-83595.

16. W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. AntiPatterns: Refactoring Software, Architectures and Projects in Crisis. John Wiley & Sons, 1998.

17. Rich Burridge. Java Shared Data Toolkit User Guide. Sun Microsystems, Inc., October 1999.

18. R. P. Carasik and C. E. Grantham. A Case Study of CSCW in a Dispersed Organization. In CHI '88: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 61–66, New York, NY, USA, 1988. ACM Press. ISBN 0-201-14237-6.

19. David Chappell. Understanding .NET. Independent Technology Guides. Addison Wesley, 1st edition, May 2002.

20. Li-Te Cheng, Susanne Hupfer, Steven Ross, and John Patterson. Jazz: A Collaborative Application Development Environment. In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 102–103, Anaheim, California, USA, October 2003. ACM Press.

21. Neville Churcher and Carl Cerecke. GroupCRC: Exploring CSCW Support for Software Engineering. In Proceedings of the 4th Australasian Conference on Computer-Human Interaction, Hamilton, New Zealand, November 1996. IEEE Computer Society Press.

22. Neville Churcher, Warwick Irwin, and Carl Cook. Inhomogeneous Force-Directed Layout Algorithms in the Visualisation Pipeline: From Layouts to Visualisations. In Australasian Symposium on Information Visualisation, (invis.au'04), volume 35 of Conferences in Research and Practice in Information Technology, pages 43–51, Christchurch, New Zealand, 2004. ACS.

23. Alistair Cockburn. Agile Software Development. Addison-Wesley, 1st edition, December 2001.

24. Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version Control with Subversion. O'Reilly Media, 1st edition, June 2004.

25. Carl Cook. Collaborative Software Engineering: An Annotated Bibliography. Technical Report TR-COSC 02/04, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, June 2004. Work in Progress.

26. Carl Cook and Neville Churcher. A Pure-Java Group Communication Framework. Technical Report TR-COSC 02/03, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, July 2003.

27. Carl Cook and Neville Churcher. An Extensible Framework for Collaborative Software Engineering. In Deeber Azada, editor, Proceedings of the

Tenth Asia-Pacific Software Engineering Conference, pages 290– 299, Chiang Mai, Thailand, December 2003. IEEE Computer Society.

28. Carl Cook and Neville Churcher. Modelling and Measuring Collaborative Software Engineering. In Vladimir Estivill-Castro, editor, Proceedings of ACSC2005: Twenty-Eighth Australasian Computer Science Conference, volume 38 of Conferences in Research and Practice in Information Technology, pages 267–277, Newcastle, Australia, January 2005. ACS.

29. Carl Cook and Neville Churcher. Constructing Real-Time Collaborative Software Engineering Tools Using CAISE, an Architecture for Supporting Tool Development. In Vladimir Estivill-Castro and Gill Dobbie, editors, Proceedings of ACSC2006: Twenty-Ninth Australasian Computer Science Conference, volume 39 of Conferences in Research and Practice in Information Technology, Tasmania, Australia, January 2006. ACS.

30. Carl Cook, Neville Churcher, and Warwick Irwin. Towards Synchronous Collaborative Software Engineering. In Proceedings of the Eleventh Asia-Pacific Software Engineering Conference, pages 230– 239, Busan, Korea, December 2004. IEEE Computer Society.

31. Carl Cook, Neville Churcher, and Warwick Irwin. A User Evaluation of Synchronous Collaborative Software Engineering Tools. In Proceedings of the Twelfth Asia-Pacific Software Engineering Conference, pages 230–239, Taipei, Taiwan, December 2005. IEEE Computer Society.

32. Donald Cox and Saul Greenberg. Supporting Collaborative Interpretation in Distributed Groupware. In Proceedings of the ACM Conference on Computer Supported Cooperative Work, pages 289–298, Philadelphia, PA, December 2000. ACM Press.

33. Bill Curtis, Herb Krasner, and Neil Iscoe. A Field Study of the Software Design Process for Large Systems. Communications of the ACM, 31 (11):1268–1287, 1988. ISSN 0001-0782.

34. Edsger W. Dijkstra. The Humble Programmer. Communications of the ACM, 15(10):859–866, 1972. ISSN 0001-0782.

35. Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft—A Tool for Visualizing Line Oriented Software Statistics. IEEE Transactions on Software Engineering, 18(11):957–968, 1992. ISSN 0098-5589..

36. Jacky Estublier. The Adele Configuration Manager. John Wiley & Sons, Inc., New York, NY, USA, 1995. ISBN 0-471-94245-6.

37. Marin Fowler. UML Distilled: A Brief Guide To The Standard Object Modeling Language. Object Technology Series. Addison Wesley, Reading, MA, 3rd edition, 2004.

38. Martin Fowler. CruiseControl: Continuous Integration Toolkit. ThoughtWorks Incorporated, November 2005.

39. Martin Fowler and Matthew Foemmel. Continuous Integration. ThoughtWorks, Inc., October 2005. URL www.martinfowler.com/ articles.

40. Jon Froehlich and Paul Dourish. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. In 6th International Conference on Software Engineering (ICSE'04), pages 387–396, Edinburgh, Scotland, United Kingdom, May 2004. IEEE.

41. G.W. Furnas. Generalised Fisheye Views. In Proc ACM SIGCHI '86 Conference on Human Factors in Computing Systems, pages 16–23,1986.

42. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

43. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison Wesley, 1995. ISBN 0201633612.

44. Emden R Gansner and Stephen C North. An Open Graph Visualization System and its Applications to Software Engineering. Software—Practice and Experience, 30(11):1203–1233, September 1999.

45. Christopher Garrett. Software Modeling Introduction: What Do You Need from a Modeling Tool? Borland Software Corporation, May 2003. White Paper.

46. James Gosling, Bill Joy, and Guy Steele. Java Language Specification, chapter 18.1. The Java Series. Prentice Hall, 2nd edition,2000.

47. Nicholas Graham, Hugh Stewart, Authur Ryman, Reza Kopaee, and Rittu Rasouli. A World-Wide-Web Architecture for Collaborative Software Design. In Software Technology and Engineering Practice, pages 22–32, Pittsburgh, Pennsylvania, August 1999. IEEE.

48. Saul Greenberg. The 1988 Conference on Computer-Supported Cooperative Work: Trip Report. In SIGCHI Bulletin, volume 20 of 5, pages 49–55. ACM, July 1989. Also published in Canadian Artificial Intelligence, 19, April 1989.

49. Jonathan Grudin. Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces. In D. Marca and G. Bock, editors, Groupware: Software for Computer-Supported Cooperative Work, pages 552–560. IEEE Press, Los Alamitos, CA, 1992.

50. Carl Gutwin, Reagan Penner, and Kevin Schneider. Group Awareness in Distributed Software Development. In CSCW ’04: Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, pages 72–81, New York, NY, USA, 2004. ACM Press. ISBN 1-58113- 810-5.

метадані

Заголовок
Підвищення семантичної ефективності моделей групової динаміки команд розробників програмного забезпечення

Автор
Бельмега Р.Я. Науковий керівник / Експерт

підрозділ
King Danylo University

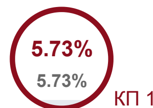
Тривога

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.

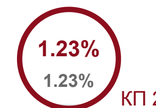
Заміна букв	ß	1
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	48

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

15930
Кількість слів

121793
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/123456789/392/%D0%9A%D0%A0%20%D0%9F%D0%B0%D1%88%D0%BD%D0%B8%D0%BA%20%D0%90.%20%D0%9F..pdf?sequence=1	66	0.41 %
2	http://repository.ukd.edu.ua/bitstream/handle/123456789/392/%D0%9A%D0%A0%20%D0%9F%D0%B0%D1%88%D0%BD%D0%B8%D0%BA%20%D0%90.%20%D0%9F..pdf?sequence=1	43	0.27 %
3	http://repository.ukd.edu.ua/bitstream/handle/123456789/392/%D0%9A%D0%A0%20%D0%9F%D0%B0%D1%88%D0%BD%D0%B8%D0%BA%20%D0%90.%20%D0%9F..pdf?sequence=1	35	0.22 %