ЗАКЛАД ВИЩОЇ ОСВІТИ

УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Чейпеш Анжеліка Степанівна

УДК 004.4

Порівняльний аналіз процесу імплементації архітектурних рішень при використанням різних паттернів проєктування

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття освітнього ступеню бакалавра

Науковий керівник

к.ф-м.н., доц

Бойчук А.М.

Івано-Франківськ – 2024

ЗВО "Університет Короля Данила"
Факультет суспільних та прикладних наук
Кафедра інформаційних технологій

Освітній ступінь: «бакалавр»
Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____
« ____ » _____ 2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Чейпеш Анджеліці Степанівній

(прізвище, ім'я, по батькові)

1. Тема роботи

Порівняльний аналіз процесу імплементації архітектурних рішень при використанням різних паттернів проектування

керівник роботи:

Бойчук Андрій Михайлович, к.ф-м.н., доцент

затверджена наказом вищого навчального закладу від « __» травня 2024 року № ___/1-НВ

2. Строк подання студентом роботи 01.06.2024

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

1. *Аналіз забезпечення безпеки компонент в побудові паттернів проектування*
2. *Проектування структура моніторингу архітектур для підтримки динамічності послуг використання паттернів проектування*
3. *Реалізація моделі системи моніторингу в контексті фреймворку OSGi*
4. *Реалізація рівнів безпечної імплементації служб і паттернів в архітектурі ПЗ*

4. Дата видачі завдання 10.02.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Аналіз забезпечення безпеки компонент в побудові паттернів проектування	10.02.2024	Виконано
2.	Проектування структура моніторингу архітектур для підтримки динамічності послуг використання паттернів проектування	25.02.2024	Виконано
3.	Реалізація моделі системи моніторингу в контексті фреймворку OSGi	10.03.2024	Виконано
4.	Реалізація рівнів безпечної імплементації служб і паттернів в архітектурі ПЗ	24.03.2024	Виконано
5.	Формування висновків	25.04.2024	Виконано
6.	Оформлення пояснювальної записки	22.05.2024	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	12.06.2024	Виконано

Студент _____ Чейпеш А.С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Бойчук А.М.
(підпис) (прізвище та ініціали)

Вихідні дані:

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

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

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу

Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

АНОТАЦІЯ

Кваліфікаційна робота присвячена виконанню порівняльного аналізу процесу імплементації архітектурних рішень при використанням різних паттернів проектування та побудові моделі інструменту динамічної верифікації, що враховує динамічні примітиви архітектури програмного забезпечення

Виконано порівняльний аналіз архітектурних рішень, що ґрунтуються на паттернах проектування та опис архітектурної моделі інструменту динамічної верифікації під час виконання та враховує деякі динамічні примітиви архітектури програмного забезпечення.

Основним результатом є порівняльний аналіз різних архітектурних рішень на основі паттернів. Це може включати ідентифікацію сильних та слабких сторін конкретних рішень, визначення найефективніших паттернів для певних вимог, а також рекомендації для вибору паттернів при проектуванні програмних систем.

Запропоновано підхід динамічного моніторингу для динамічної системи на основі архітектури SOA під час виконання у відкритому середовищі. Цей підхід динамічного моніторингу вставляє елементів в момент прив’язки клієнт-сервер, а не «статично» під час компіляції чи завантаження при компонуванні архітектури на основі паттернів.

КЛЮЧОВІ СЛОВА: ВЕБ-СЛУЖБА, ПАТТЕРН, ТЕХНОЛОГІЯ, СИСТЕМА МОНІТОРИНГУ, САМОВІДНОВЛЮВАНІ СИСТЕМИ, СПЕЦИФІКАЦІЯ, ВЛАСТИВІСТЬ, АДАПТОВАНІСТЬ, ПРОЕКТУВАННЯ СИСТЕМИ

ANNOTATION

The qualification work is devoted to performing a comparative analysis of the implementation process of architectural solutions using various design patterns and building a model of a dynamic verification tool that takes into account the dynamic primitives of the software architecture

A comparative analysis of architectural solutions based on design patterns and a description of the architectural model of the dynamic verification tool at runtime and taking into account some dynamic primitives of the software architecture are performed.

The main result is a comparative analysis of various architectural solutions based on patterns. This may include identifying the strengths and weaknesses of specific solutions, determining the most effective patterns for certain requirements, and making recommendations for choosing patterns when designing software systems.

A dynamic monitoring approach is proposed for a dynamic system based on SOA architecture at runtime in an open environment. This dynamic monitoring approach inserts elements at the time of client-server binding, rather than "statically" at compile or load time when composing a pattern-based architecture.

KEY WORDS: WEB SERVICE, PATTERN, TECHNOLOGY, MONITORING SYSTEM, SELF-HEALING SYSTEMS, SPECIFICATION, PROPERTY, ADAPTABILITY, SYSTEM DESIGN

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AOP - Aspect-Oriented Programming

DSL - Domain-Specific Language

D-SOA - Dynamic Service-Oriented Architecture

EJB - Enterprise Jave Bean

FSM - Finite State Machine

IPOJO - Inject Plain Old Java Objects

IDS - Intrusion Detect Systems

JavaMOP - Java Monitoring-oriented Programming

JBI - Java Business Integration

JML - Java Modeling Language

JSON - JavaScript Object Notation

JSON-WSP - JavaScript Object NotationWeb-Service Protocol

JVM - Java Virtual Machine

LTL - Linear temporal logic

OSGi - Open Services Gateway initiative

PTLTL - Past Time Linear Temporal Logic

PVS - Property Verification System

REST - Presentational State Transfer

SOA - Service-Oriented Architecture

SSU Safe Service Usage

STM Software Transactional Memory

TM Transactional Memory

UDDI Universal Description, Discovery and Integration

WSDL Web Services Description Language

ВСТУП

Актуальність теми. В контексті швидких технологічних змін та зростаючої складності програмного ландшафту, дослідження порівняльного аналізу архітектурних рішень на основі паттернів проектування залишається вкрай актуальним і важливим для подальшого розвитку програмної індустрії.

Швидкі технологічні зміни - сучасний індустріальний ландшафт визначається швидкими технологічними змінами та зростаючою складністю програмних систем. Вивчення та оптимізація архітектурних рішень, заснованих на паттернах, є критично важливим для ефективного реагування на ці зміни.

Зростання розмірів та складності програмних продуктів - зростом розмірів та складності програмних продуктів виникає необхідність вдосконалення архітектурних рішень. Порівняльний аналіз паттернів проектування може допомогти вибрати оптимальні стратегії для розробки та підтримки великих проектів.

Потреба у високій якості програмного коду - збільшення уваги до якості програмного коду та стійкості архітектури зумовлено необхідністю мінімізації помилок, підвищення ефективності та покращення зручності розробки.

Розвиток підходів до архітектурного проектування - урахуванням змін у розумінні архітектурних паттернів та з'явлення нових підходів, актуальність дослідження полягає в оновленні та розширенні знань у цій області.

Забезпечення високої ефективності та конкурентоспроможності - для компаній та розробників важливо мати доступ до оптимальних архітектурних рішень для забезпечення високої ефективності, конкурентоспроможності та швидкого впровадження нових функцій у програмні продукти.

Порівняння роботи з відомими розв’язаннями проблеми

Порівняння роботи з відомими розв'язаннями проблеми у вашому дослідженні допоможе визначити переваги та недоліки обраного підходу та надати контекст для отриманих результатів. Нижче розглядаються кілька аспектів порівняння:

- Ефективність та продуктивність;
- Гнучкість та розширюваність;
- Стабільність та надійність;
- Зручність використання та розробки

Метою роботи є проведення порівняльного аналізу архітектурних рішень, що ґрунтуються на паттернах проектування та описі архітектурної моделі інструменту динамічної верифікації під час виконання та враховує деякі динамічні примітиви архітектури програмного забезпечення.

Досягнення мети включало розв'язання таких **задач**:

1) Аналіз забезпечення безпеки компонент в побудові паттернів проектування;
2) Проектування структура моніторингу архітектур для підтримки динамічності послуг використання паттернів проектування;
3) Реалізація моделі системи моніторингу в контексті фреймворку OSGi;
4) Реалізація рівнів безпечної імплементації служб і паттернів в архітектурі ПЗ.

Об'єкт дослідження: архітектурні рішення, що базуються на паттернах проектування

Предметом дослідження є моделі та методи аналізу архітектурних рішень систем моніторингу на основі паттернів

Методи дослідження

В роботі застосовано застосовано аналіз літературних джерел, емпіричні методи, методи експертного опитування, методи порівняльного аналізу.

Результати роботи. Основним результатом кваліфікаційної роботи є: системний аналіз різних архітектурних рішень на основі паттернів, що може включати ідентифікацію сильних та слабких сторін конкретних рішень, визначення найефективніших паттернів для певних вимог, а також рекомендації для вибору паттернів при проектуванні програмних систем. Наукова новизна включає в себе три аспекти – порівняльний аналіз архітектурних рішень, розробку критеріїв порівняння та створення та використання нових критеріїв для

порівняльного аналізу дозволяє вперше систематизувати та оцінювати архітектурні рішення з точки зору їх відповідності вимогам та якості

Структура роботи. Розділи – 3. Загальний обсяг основної частини – 83 сторінки. Список використаних джерел – 43.

РОЗДІЛ 1. Теоретичні основи та аналіз забезпечення безпеки динамічних компонент при побудові паттернів проектування архітектури

1.1 Огляд динамічної сервіс-орієнтованої архітектури

SOA складається з великої кількості автономних і самодостатніх сервісів. Кожна функція програми є послугою. Сервіс в архітектурі на основі SOA містить інтерфейс сервісу, реалізацію сервісу та контракт на обслуговування. Служба в інтерфейсі розкриває абстрактну функціональність послуг. Реалізація служби забезпечує базову бізнес-логіку та дані для виконання вказаних функцій в інтерфейсі служби. Договір про надання послуг визначає функціональні можливості служби , тип обов'язкового протоколу та обмеження для обслуговування клієнта; він також базується на стандартах і не залежить від платформи та зберігається в репозиторії послуг.

Архітектура динамічної SOA (D-SOA) складається з динамічних і слабко пов'язаних служб. Життєвими циклами служб можна динамічно керувати дистанційно під час виконання через слабкий зв'язок між клієнтом і службою. Наприклад, служби можуть з'являтися та зникати динамічно на регулярній основі, не впливаючи на роботу інших служб. Фреймворк D-SOA має деякі правила для інформування відповідної служби про змінений стан життєвого циклу служби (початок або зупинка) або допомоги клієнту в пошуку більш підходящої реалізації служби, ніж поточна.

Оскільки служби не пов'язані між собою та слабко пов'язані, взаємодія служб дає змогу службі на стороні виклику запитувати деякі функції сервера через репозиторій, який розкриває відповідні контракти. Згодом служба на стороні виклику прив'язується до служби, і їй дозволяється викликати методи через інтерфейс служби, якщо типи контрактів збігаються. Крім того, деякі

служби можна об’єднати разом, щоб створити нову службу з різними функціями під час виконання та вийти на новий рівень деталізації.

Завдяки цим характеристикам (слабко пов’язані, багаторазові, рекомпонування з різними рівнями деталізації) SOA привертає все більше уваги великих компаній.

і широкі території. Для прикладу ми можемо навести: систему RFID на основі SOA, область управління радіоактивними відходами, поле інтелектуального аналізу даних, управління біомедичними даними, хмарні обчислення. Існують також різні підходи до реалізації SOA серед двох сімейств: веб-сервіси та інші більш локальні підходи, такі як OSGi [ ].

У цій роботі ми зосереджуємось на фреймворку OSGi. Зазвичай він використовується в системах 24/7 , де система не перезапускається, коли служба з’являється або зникає. Ця структура призначена для вбудованих систем, таких як автомобілі, ADSL-блоки або мережеві системи. У таких системах веб-сервіси не можна використовувати через відсутність підключення, обмежену пропускну здатність мережі або з причин ефективності.

Сервісно-орієнтована архітектура (SOA) зосереджена на слабкому зв’язку клієнт-сервер через публічні інтерфейси. Клієнт зазвичай запитує доступ до служби через репозиторій. Після цього клієнт прив’язується до служби, і йому дозволяється викликати методи, якщо типи інтерфейсу збігаються. У динамічній SOA кожен виклик служби слід розглядати як повне перемикання контексту, оскільки потенційно нові служби можуть з’явитися, а інші зникнути під час виконання, навіть якщо ці служби є станом. Ця динамічна діяльність повинна мати якомога менше наслідків на стороні клієнта.

З точки зору динамічної SOA, вирішення слабких зв’язків і динамічних проблем служб сьогодні є справжньою проблемою. По-перше, зв’язування клієнта зі службою є питанням відповідності інтерфейсу через слабкий зв’язок служб, але ні клієнт, ні служба не мають жодної гарантії, що інша частина поводиться як очікувалося. По-друге, кожна система, що реалізує динамічну SOA, стикається з проблемою застарілих посилань, спричиненою мобільністю

сервісів. Оскільки застаріле посилання на службу потенційно може призвести до «нульового посилання на вказівник» або до неправильного результату, це може призвести до збою системи.

Метою цієї роботи є не лише визначення того, чи є поведінка клієнт авторизованою чи ні в динамічній системі SOA. Це також покращує відмовостійкість динамічної системи SOA під час зникнення служби. І останнє, але не менш важливе, у цій тезі всі служби можна розглядати як служби з підтримкою стану в системі такого типу. Для досягнення цих цілей ми перевіряємо два випадки:

По-перше, важливо постійно забезпечувати автентичність клієнтів і валідність дій, які виконуються після зіставлення інтерфейсу для більшості систем. Кожного разу, коли клієнт надсилає запит серверу, можна перевірити формально визначене обмеження, щоб переконатися, що клієнт має право виконувати цей виклик. Отже, система моніторингу часу виконання

можна використовувати для перевірки такої поведінки в системах D-SOA. Існують деякі традиційні підходи до моніторингу виконання для перевірки конкретної поведінки клієнтського доступу до служби. Ці підходи включають статичне відображення та моніторинг служб, але існує обмеження цих моніторів, коли служба зникає або динамічно з’являється нова, ці монітори не можуть продовжувати моніторинг нової заміненої служби без перезапуску системи. Ця робота визначає монітор часу виконання зі стійкістю до динамічності та комплексністю для динамічної SOA. У світлі такої мети ми досліджуємо можливість постійного моніторингу запитів нових послуг від клієнтів без перезавантаження системи.

По-друге, з огляду на дійсні посилання на стан послуг у динамічній SOA-системі важливо мати справу з динамічністю послуг. Посилання на послугу — це визначений покажчик клієнта, отриманий для використання його служби в цій системі. Отже, ми можемо запропонувати деякі інструменти на стороні клієнта, щоб допомогти запущеній динамічній системі SOA. Коли служба зникає , її система все ще може працювати або викликати виключення. Ми можемо

використовувати ці інструменти, щоб додати кілька кодів на стороні клієнта для отримання нового посилання на службу, щойно нова служба стане доступною замість зниклої. Клієнту не потрібно перезапускати після цієї заміни служби, і це також дозволяє уникнути використання застарілих посилань.

У цій роботі основні внески перераховані таким чином:

Запропоновано підхід динамічного моніторингу для моніторингу динамічної системи SOA під час виконання у відкритому середовищі:

- Цей підхід динамічного моніторингу вставляє монітори в момент прив’язки клієнт-сервер, а не «статично» під час компіляції чи завантаження. Цей підхід може здійснювати динамічні відображення від монітора до служби чи методу під час виконання , навіть якщо служби з’являються чи зникають, оскільки монітор має той самий життєвий цикл із інтерфейсом служби, що контролюється, а не реалізацією служби ;
- Цей тип монітора може перевіряти поведінку клієнтів, які використовують послуги, а інша поведінка, пов’язана з цим сервісом, не може обійти спостереження монітора;
- Опис властивостей цього монітора складається з властивості сторони інтерфейсу та властивості сторони клієнта. Ці властивості цього монітора можуть відповідним чином перевіряти поведінку кожного клієнта, який -використовує службу через його контрольований інтерфейс з кожним ідентифікатором клієнта. Властивість сторони інтерфейсу — це вхід монітора;
- Реалізація цього монітора реалізована системою OSGiLarva, яка описує -події виклику методів, а також події фреймворку OSGiLarva;
- Система моніторингу також може контролювати складну систему з кількома сервісними інтерфейсами та перевіряти атомарність використання сервісів. Ці властивості інтерфейсів можна описати в контексті "глобальних" відповідно. Вони відрізняються назвою інтерфейсу.
- Цей монітор генерує записи та виводить користувачам або менеджерам під час виконання, які можуть вжити деяких необхідних заходів до контрольованої програмної системи під час досягнення певного стану.

Пропонується рівень «безпечного використання сервісу» на стороні клієнта для підвищення здатності до самовідновлення використання сервісу в динамічній системі SOA.

- Цей рівень знає про застарілі посилання. Під час виконання клієнтам потрібні два кроки, щоб запобігти використанню застарілих посилань, не вимагаючи перезапуску клієнтів і не змінюючи зовнішні служби: якщо є нова служба для заміни зниклої служби, автоматично виконується заміна служби для клієнтів, інакше надсилається виключення застарілих посилань для клієнтів.
- Цей рівень використовує транзакційний підхід для забезпечення узгодженого використання служби під час виконання. Коли використовується зникла служба, блок виконання виконує відкат і повертає всі значення параметрів, пов’язаних із виконуваними методами в ньому.

Нарешті, пропонується інша архітектура динамічного моніторингу, яка об’єднує запропоновану систему OSGiLarva та рівень SSU. Він використовується для моніторингу безпечного використання служб і запобігання використанню застарілих посилань динамічної системи SOA у відкритому середовищі.

Ця запропонована архітектура моніторингу під назвою NewMS компенсує відсутність системи OSGiLarva трьома способами:

- Він усвідомлює використання застарілих посилань і обробляє це на рівні SSU.
- Це дозволяє більш точно виражати властивості. Наприклад, можна розглянути процедуру обробки застарілих посилань.

1.2 Дослідження веб-сервісів а архітектрі SOA

Веб-сервіси є однією з реалізацій SOA. Це частина коду, доступна в мережі, з властивостями, які повинні бути самоописаними та самодостатніми. Він підтримує взаємодію між різними машинами з конкретними бізнес-функціями в мережі .

Як показано на рис. 1.1 , існує три ролі у структурі веб-сервісу: постачальник послуг, репозиторій послуг і споживач послуг. Постачальник послуг надає впровадження послуг для реалізації визначених інтерфейсів послуг. Різні постачальники послуг можуть розробляти різні реалізації для одного інтерфейсу служби для підтримки швидкого оновлення служби. Ці реалізації послуг незалежні. Роль сховища послуг (наприклад, інтеграція та розгортання послуг) полягає в управлінні всіма послугами від постачальників послуг. Споживач послуг може надсилати повідомлення, щоб знайти послугу в сховищі послуг. Якщо споживач отримує відповідь щодо запитуваної послуги, він може зв’язатися з цією запитуваною реалізацією послуги від постачальника послуг і може використовувати її. Реалізація конкретної послуги прозора для споживача послуги.

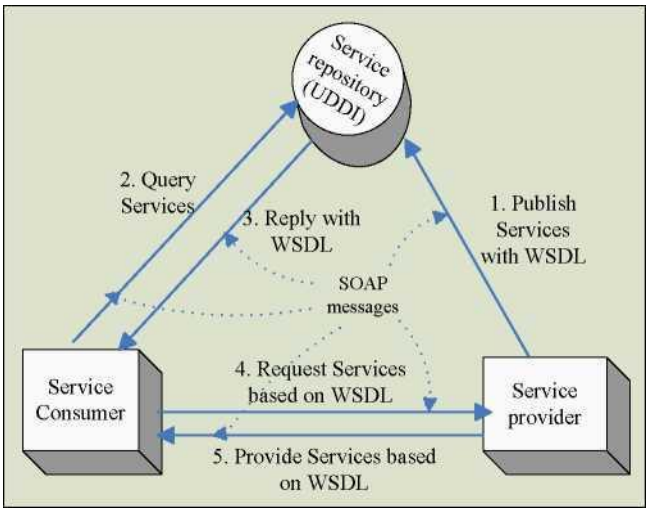


Рисунок 1.1 - Ролі та взаємодії в веб-сервісі XML для реалізації SOA

Існувало багато мов розмітки, які використовувалися для розробки веб-сервісів, наприклад : JSON, JSON-WSP, REST і RESTful, XML-SOAP. JavaScript Object Notation (JSON), яка є легкою мовою розмітки для обміну даними в Інтернеті. JSON-WSP (JavaScript Object Notation Web-Service Protocol) – це веб-протокол служби через JSON для опису, запиту та відповіді на служби. Representational State Transfer (REST) – це набір обмежень і правил архітектури, що застосовуються до розробки веб-сервісів. Це також орієнтована на ресурси

архітектура. Веб-API RESTful – це веб-API, реалізований за допомогою принципів HTTP і REST. Стандарт XML-SOAP використовується для опису, публікації, пошуку, зіставлення та налаштування веб-служб. У цій роботі ми пропонуємо зосередитися на найбільш використовуваному підході: XML-SOAP. Потім ми пояснимо, як використовувати його у веб-службах.

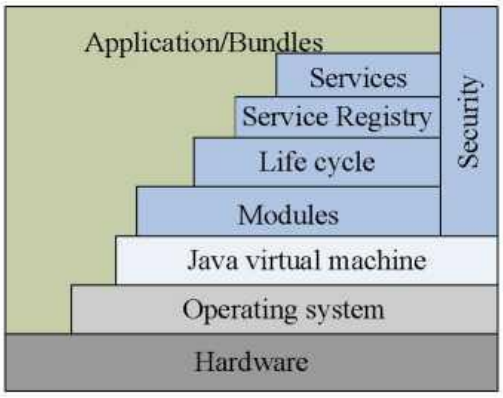
Мова опису веб-сервісів (WSDL), простий протокол доступу до об'єктів (SOAP) і сервіси універсального опису, виявлення та інтеграції (UDDI) є трьома ключовими елементами платформи в архітектурі веб-сервісу XML-SOAP, які також представлені на рис. 1.1 . WSDL і SOAP описані на основі формату XML. WSDL [3] використовується для опису інформації про веб-сервіс: тип транспорту (наприклад, SOAP), методи інтерфейсу веб-сервісу, параметри та URI веб-сервісу. Він використовується для публікації та запиту послуг. SOAP [4] – це механізм передачі послуг в архітектурі веб-служб. Він використовується для обміну інформацією про структуру веб-служби з іншими системами через HTTP. Це дозволяє уникнути перетворення інформації між різними протоколами. UDDI — центр реєстру послуг Universal Description, Discovery та Integration. Використовується для реєстрації нових служб за допомогою файлу WSDL через протоколи SOAP/HTTP. Це як жовта сторінка файлів WSDL. Споживачі послуг можуть знаходити зареєстровані служби з файлами WSDL з UDDI через протокол SOAP/HTTP у гетерогенних і розподілених середовищах .

Нарешті, веб-сервіси розглядаються як угоди між внутрішніми та зовнішніми підприємствами, бізнесами B2B та B2C тощо. Наприклад, у [29] запропоновано структуру, засновану на концепції SOA та технології веб-сервісу, для керування -системою управління записами про пакування радіоактивних відходів із трьома рівнями. У [20] інтерактивна функціональність застарілих систем представлена як веб-сервіс за допомогою підходу до системи, заснованої на SOA. Це рішення дозволило цим застарілим і різнорідним системам стати взаємопов'язаними та сумісними через мережу. У автори запропонували службу інтелектуального аналізу даних з алгоритмами інтелектуального аналізу даних. Цей тип служби розглядається як веб-служба для неспеціалістів у видобутку

даних у SOA. У [9] веб-сервіси розроблені на основі послуг охорони здоров'я в системі SOA. Така система охорони здоров'я може підвищити якість прийняття рішень і своєчасне генерування попереджень для лікарів, доглядальників і людей похилого віку.

Специфікація платформи послуг OSGi була створена альянсом OSGi. Вона визначає модель управління життєвим циклом програми Java, розміщеної на віртуальній машині [9]. Він має деякі API для керування життєвим циклом компонентів програмного забезпечення з будь-якого місця в мережі. Платформа дозволяє дистанційно завантажувати та динамічно розгортати додатки за своєю відкритою специфікацією у своєму середовищі, залишаючись незалежною від системи, на якій вона встановлена. Служби OSGi можуть працювати на різних пристроях від дуже маленьких до дуже великих. Різні споживачі послуг, постачальники, розробники, постачальники можуть добре працювати разом на цій специфікації платформи. Ця структура реалізує повну та динамічну модель компонентів на основі багаторівневої архітектури.

Ця структура складається з шести основних шарів, як показано на рис. 1.2



.

Рисунок 1.2 - Фреймворк OSGi

1. Рівень пакету: пакет — це основна концепція платформи OSGi. Це єдина одиниця модуляризації, яка складається з набору класів Java (пакетів, служб), файлів конфігурації та інших ресурсів (наприклад, зображень, звуків тощо). Цей шар буде працювати з усіма іншими шарами. У кожному пакеті є принаймні два

методи: BundleActivator.start (BundleContext) і BundleActivator.stop (BundleContext). Якщо фреймворку потрібно запустити цей пакет, потрібно викликати попередній метод. Цей метод використовується для реєстрації служб або для призначення будь-яких ресурсів, необхідних для цього пакету. Останній метод викликається, коли фреймворку потрібно зупинити цей пакет . Коли цей пакет зупинено, цей пакет не може викликати жодних об’єктів каркаса, і його не може викликати жодний пакет b, доки він не запуститься знову.

2. Сервісний рівень: пропонує набір функціональних можливостей для публікації, виявлення та прив’язки до об’єктів Java, а також сповіщення про зміни, які відбуваються в службах у середовищі. Служба — це звичайний об’єкт Java, який зареєстровано в одному або кількох інтерфейсах Java на рівні реєстру послуг

Фреймворк OSGi. Послуга — це рішення, яке пропонує платформа, щоб уникнути тісної прив’язки між компонентами. Зв’язування можна виконати за допомогою посилання на службу замість самого об’єкта служби.

3. Рівень реєстру послуг: API цього рівня використовується для керування службами щодо реєстрації послуг , відстеження послуг і генерації довідкових даних про служби . Пакет використовується для реєстрації об'єкта шляхом реєстрації Сервісу. Служби клієнта шукають відповідні об’єкти в реєстрі послуг із посиланням на службу. Коли інструмент відстеження послуг - використовується в цьому реєстрі послуг, він може прослуховувати ServiceEvents відстежуваної служби (наприклад, Unregistering, Registered, Modified і Modified_ Endmatch), а також отримання та звільнення служби.

4. Рівень життєвого циклу: керування життєвим циклом пакетів, надане фреймворком OSGi, оскільки деякі API можуть дистанційно керувати запуском, зупинкою, оновленням, встановленням і видаленням пакетів без необхідності перезавантаження. Звичайний життєвий цикл комплекту показано на рис. 1.3 . Запуск і зупинка є середніми станами в життєвому циклі комплекту OSGi. Наприклад, коли виконується команда «старт», стан пакету переходить із «Вирішено» в «Активний». При виконанні команди "стоп" стан пакету

змінюється з "Активний" на "Вирішено". Він забезпечує дистанційне керування пакетами з динамічністю.

5. Рівень модулів: на цьому рівні для Java визначено модуль модуляції. Модуль модуляризації визначає інкапсуляцію та декларацію взаємозв'язків залежності між пакетами: як пакет може імпортувати та експортувати код? Який порядок між пакетами експорту та імпорту?.

6. Рівень віртуальної машини Java: він керує завантаженням класів Java для кількох пакетів. У локальній структурі OSGi кілька пакетів працюють в одній JVM для спільного використання пакетів і координації з іншими пакетами.

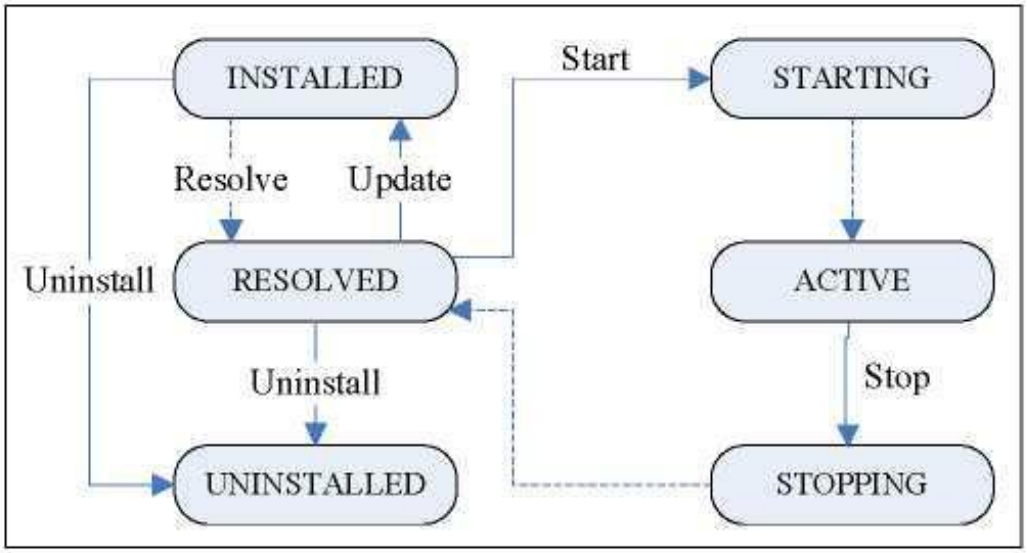


Рисунок 1.3 - Життєвий цикл OSGi Bundle

З вищевказаного вступу та розгляду кожної колекції шарів ми знаємо, що фреймворк OSGi з його реєстром послуг забезпечує легку модель для публікації, пошуку та зв'язування служб у своїй JVM. Ця структура підтримує характеристики сервіс-орієнтованої архітектури. Рівень життєвого циклу надає API для пакетів для керування службами на рівні модуля. Ці характеристики дозволяють цій структурі стати динамічним підходом SOA. Широко використовується платформа OSGi Service: Домашня автоматизація на базі платформи OSGi; Автомобільна промисловість прийняла платформу OSGi для підтримки різних послуг виробників транспортних засобів. Крім того, він

підтримує службу дистанційного виклику безпілотних транспортних засобів [26]; Настільні ПК, сервери (сервери високого класу, включаючи мейнфрейми), Nokia і Motorola запровадили стандарт технології OSGi для наступного покоління смартфонів.

Архітектура веб-сервісу є популярним підходом до реалізації SOA через мережу , тоді як OSGi є моделлю динамічних компонентів на основі динамічної SOA. Ось основні відмінності між архітектурою веб-сервісів і фреймворком OSGi:

1. Перегляд служби: архітектура веб-служб, як правило, не може мати повне уявлення про систему, тобто можна спостерігати або за клієнтом, або за сервером, але не за обома. Фреймворк OSGi може скласти повну картину, також беручи до уваги події фреймворку OSGi, такі як реєстрація послуг, запити на обслуговування різними клієнтами тощо. Це можливо, оскільки фреймворк OSGi забезпечує віддалене та динамічне керування життєвим циклом.

2. Швидкість обміну повідомленнями служби: існують різні механізми транспортування служби в обох підходах. Локальні служби OSGi спілкуються один з одним так само, як і звичайні виклики Java. Усі веб-служби спілкуються одна з одною за допомогою прив’язки SOAP із протоколами HTTP/TCP/UDP. Службові методи OSGi викликаються в тисячі разів швидше, ніж виклики веб-служб.

3. Служба зникає під час виконання: фреймворк OSGi уникає «нульового посилального покажчика», пов’язаного зі зникненням служби, без використання «Сервісного відстеження». Коли послуга була завантажена споживачем служби, цей споживач служби може викликати свої методи служби після скасування реєстрації. Але для веб-сервісів цей виклик індукує нульовий посилальний покажчик під час виконання.

4. Враховуючи вартість: усі локальні пакети OSGi працюють в одній JVM для спільного використання та координації з іншими пакетами. Це мінімізує обсяг пам’яті та підвищує продуктивність. Завдяки цьому він забезпечує майже нульову вартість зв’язку між додатками, представленого в [8].

1.3 Дослідження технології AspectJ. Системи моніторингу

Існує кілька близьких робіт із використанням технології AspectJ. Серед них монітор виконання, Larva, JavaMOP і система динамічного моніторингу . Отже, ми коротко представляємо технологію AspectJ у фоновому режимі.

Аспектно-орієнтоване програмування (АОР) є парадигмою програмування. Він використовує наскрізний підхід, щоб отримати загальну поведінку з внутрішньої сторони упакованих об’єктів, а потім інкапсулювати загальну поведінку в модель багаторазового використання, яка називається рect. Загальна поведінка впливає на кілька класів і відрізняється від бізнес-процесів об’єктів, таких як ідентифікація повноважень, журналювання, процес транзакцій тощо. Аспект дозволяє зменшити кількість повторюваних кодів, полегшити зв’язок між модулями та підвищити працездатність і зручність обслуговування в програмній системі.

Реалізації АОР мають деякі вирази аспектів, які можуть інкапсулювати наскрізні проблеми програмних систем. AspectJ, яка є найбільш універсально використовуваною мовою АОП, є бездоганним аспектно-орієнтованим розширенням мови програмування Java. У ньому є деякі вирази для інкапсуляції наскрізних проблем в аспект, наприклад спільна точка, pointcut, рада, міжтипове оголошення. Спільна точка — це метод класу з оригінальної системи. Це абстрактне поняття в АОР, його не потрібно визначати. Точковий розріз — це структура для захоплення вказаного набору точок з’єднання. Він просто створює посилання на цільову систему для спостереження. Порада вказує код виконання точкового розрізу. Він може надати конкретну логіку виконання за допомогою спеціальної обробки: до, після та навколо. Визначений точковий розріз буде виконано до , після або навколо захопленої точки з’єднання (наприклад, методи класу). Оголошення між типами застосовується для оголошення наскрізних класів та їх ієрархій. Тому pointcut і ad vice динамічно обробляють потік програми під час виконання, оголошення між типами виконується під час розробки. Аспект

інкапсулює ці вирази аспектів, щоб сформувати чітку модульність наскрізних проблем. Аспект можна відокремити від цільової системи та повторно використовувати, наприклад, для перевірки помилок , моніторингу, журналювання тощо.

Після узгодження інтерфейсу служби важко гарантувати безпечне використання компонентів у системах D-SOA. Якщо ми використовуємо класичний інструмент моніторингу для перевірки та перевірки деяких конфіденційних дій під час виконання системи D-SOA, зникнення або поява служби призведе до небажаних речей для класичного інструменту моніторингу, таких як втрата інформації, зникнення монітора тощо. Отже, дві характеристики, які ми вважаємо важливими в інструменті моніторингу для перевірки системи D-SOA: стійкість до динамічності та комплексність моніторингу.

- стійкість до динамічності: це стосується збереження потоку поведінки. У разі заміни служби, що контролюється, монітор і його стан мають бути передані , що означає, що властивість, що контролюється, не може бути жорстко пов’язана з кодом.
- повнота моніторингу: це означає, що ми не можемо дозволити службам обмежувати те, що може спостерігати монітор. Якщо ми хочемо перевірити майно, нам потрібно переконатися, що всі відповідні події відстежуються.

Ми пропонуємо класифікувати існуючі підходи до верифікації під час виконання відповідно до конфігурації монітора по відношенню до контрольованих програмних систем. Властивість, що контролюється, може бути: написана вручну всередині коду, автоматично введена всередину коду, виключена з коду і моніторинг веб- сервіс. Для аналізу стійкості до динамічності та моніторингу всеосяжності в кожній із цих родин, спочатку ми повинні дати пояснення щодо деяких стилів опису властивості.

Програмування, орієнтоване на моніторинг Java (JavaMOP) [24, 25] аналізує структуру, присвячену моніторингу програмування Java, яка приймає деякі незалежні формалізми специфікації. Він спрямований на зменшення

розриву між формальною специфікацією та реалізацією шляхом інтеграції їх у свою оригінальну систему. Його можна використовувати для розробки монітора виконання для розробки програмного забезпечення надійності, безпеки та надійності. Його можна використовувати для розробки логіки подій (наприклад, FSM, PTLTL, LTL тощо) у формальній специфікації проти програмних реалізацій. Розроблена специфікація компілюється JavaMOP як код AspectJ, а потім вплітається в цільову систему реалізації будь-яким компілятором AspectJ (наприклад, ajc).

Опис властивостей JavaMOP може виражати поведінкові властивості та властивості живучості за допомогою LTL і PTLTL. Однак LTL-виражена жива властивість буде згенерована технологією aspectj. Згенеровані властивості моніторингу складаються з кількох формул FSM. Опис властивостей моніторингу не покращує виразність, за винятком того, що його легше написати.

Larva [32] — це інструмент, який вставляє код моніторингу в програму Java для перевірки описаної властивості у файлі сценарію Larva. Цей інструмент, який об’єднує перехоплення викликів за допомогою методів аспектно-орієнтованого програмування, закритий для JavaMOP. Обидва вони дозволяють відстежувати деякі поведінкові властивості, але властивості в реальному часі можуть бути виражені лише в Larva. До речі, він може не тільки описувати конкретні методи обслуговування, а й контролювати певні динамічні події, що відбуваються за допомогою таймерів.

Коли програмне забезпечення, що контролюється, запускається з системою Larva (рис. 1.5), його сценарій властивостей компілюється компілятором Larva. Компілятор Larva генерує два основні результати зі свого сценарію:

1. Аспектно-орієнтований код: цей код, який пов’язує код моніторингу з програмним забезпеченням, що контролюється, яке має на меті витягти події, що контролюються. Він буде статично впроваджувати деякі виклики програмного забезпечення, що контролюється, за допомогою компілятора Aspect] під час кодування, під час компіляції або під час завантаження.

2. Код класу Java: цей код використовується для перевірки відповідності вилучених подій розробленій властивості. Система перевірки знаходиться за межами контрольованого програмного забезпечення. Після перевірки розробленої події система перевірки надсилає відстежувані записи користувачам. Користувачі мають виконати деякі необхідні дії з цільовою системою, коли виводиться контрольований запис.

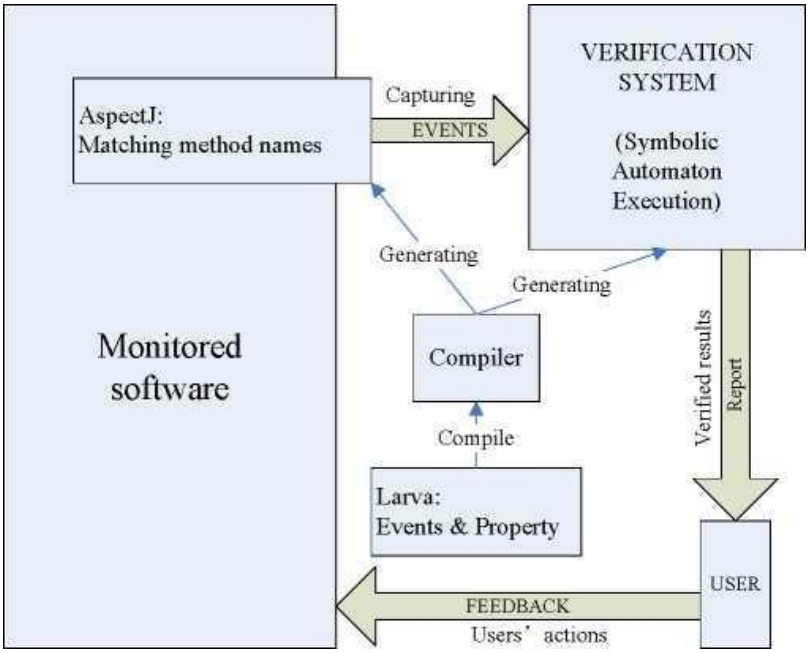


Рисунок 1.5 - Система Larva в програмній системі

Опис властивостей личинки може виражати поведінкові властивості та тимчасові властивості. Його сутність полягає в методах (внутрішніх і зовнішніх). У Larva та JavaMOP є дійсно близькі характеристики в них. Однак Larva показав кращі результати щодо споживаних ресурсів, ніж JavaMOP.

1.4 Моніторинг веб-служб та процес агностичного кодування

Існує ряд робіт, наприклад, [13, 14], які підтримують моніторинг веб-сервісів. Для цього запропоновано структуру динамічного моніторингу з моделлю сценарію моніторингу та інструментальним рівнем для моніторингу під

час виконання систем на основі середовища виконання SOA. У цьому підході використовується інструментарій АОП. У кожну розкриту службу введено код сокета-перехоплювача, який обгортає його сокетом. Кожен перехоплювач сприймається як послуга та публікується з його інтересом і пріоритетом. Після реєстрації перехоплювача ця реєстраційна інформація буде повідомлена кожному сокету перехоплювача з його загорнутою службою для порівняння атрибутів перехоплювача з атрибутами сокета. Якщо атрибути збігаються, цей перехоплювач додається в чергу відповідних сокетів за своїм пріоритетом. Тоді введений монітор може почати відстежувати відповідний виклик служби. Також є деякі недоліки:

• Код сокета перехоплювача потрібно впроваджувати в кожну розкриту службу, сокет обгортається цією службою, навіть вставляючи в деякі служби, які повністю не потребують моніторингу.

• Зараз цей монітор зосереджується лише на виклику служби, а не на конкретних параметрах виклику чи реалізованій бізнес-логіці. Наприклад, він може контролювати частоту викликів і частоту помилок.

Java Business Integration (JBI) — це свого роду модель веб-сервісів. Оскільки AOP може розглядати наскрізні аспекти поведінки системи якомога окремо та без примусової модифікації вихідного коду, збагачення JBI-сумісного моніторингу реалізовано за допомогою технології AspectJ. Як показано на рис. 1.6 , визначеним AspectJ pointcuts можна дозволити перетинати інтерфейси JBI. Оскільки збереження вихідного коду та файлів класів дозволяє уникнути модифікації, автори використовують час завантаження, переплітаючи ці визначені аспекти спеціальним агентом Java. Цей інструментарій моніторингу базується на специфікації AOP enrich JBI. Отже, цей монітор може бути значним обмеженням у вираженні політик безпеки. Технологію AspectJ можна використовувати для моніторингу пунктів програми за її порадами (обмежтеся такими способами: до, після, навколо), а не за логікою бізнес-процесів. Збагачення JBI-сумісного моніторингу може виражати інваріантну властивість. Його властивістю є деталізація інструкцій.

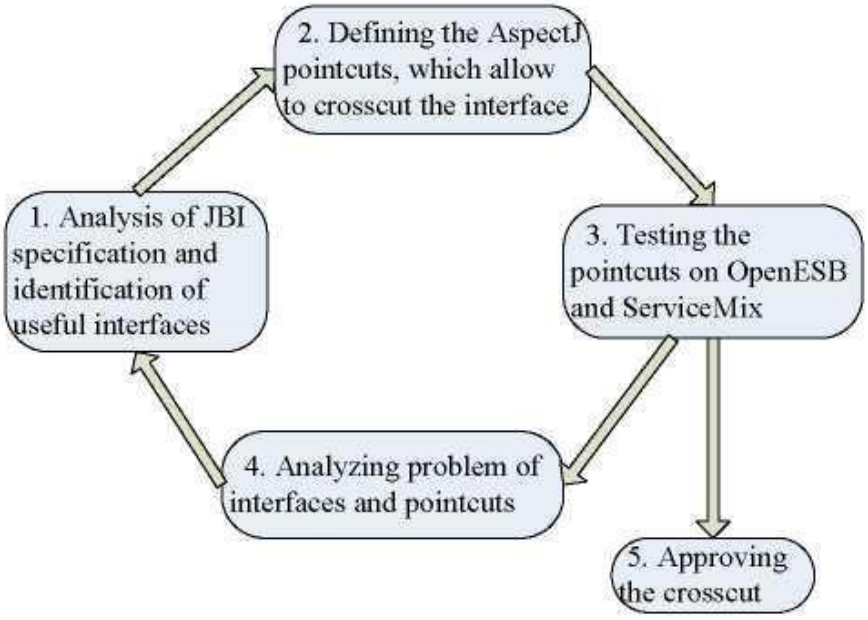


Рисунок 1.6 - Підхід до наскрізного аналізу AOP

У автори забезпечили як стійкість до динамічності, так і -характеристики комплексності (навіть якщо вони явно не визначені як такі), прослуховуючи події з механізму композиції веб-сервісу. Крім того, в [13] ця архітектура моніторингу підтримує як монітори екземплярів, так і монітори класів. Монітори екземплярів перевіряють поведінку окремого екземпляра бізнес-процесу BPEL; монітори класів витягують або збирають інформацію з перевірених дій усіх моніторів екземплярів, вони прагнуть отримати синтезовану інформацію з точки зору класу. Однак, наскільки нам відомо, подібних методів моніторингу для структури OSGi не запропоновано. Крім того, контекст не однаковий, оскільки в контексті веб-сервісу ми можемо легко розрізняти абонентів за їхньою IP-адресою та номером порту, але неможливо знати, хто телефонує, або який клас чи програмне забезпечення здійснює виклик . Інструмент моніторингу може виражати властивості поведінки, властивості живості та властивості часу. Його властивістю є деталізація логіки бізнес-процесів.

Дійсно, хоча технічно можливо використовувати AspectJ для підтримки динамічного завантаження та вивантаження класу в OSGi, тоді контрольований пакет повинен оголосити імпорт бібліотеки AspectJ у своєму файлі Manifest —

операція, яка насправді не прозора для служби. Зверніть увагу, що це обмеження не існує в реалізації Equinoxe OSGi (Eclipse). Оскільки деякі варіанти було б зроблено в конфігурації фреймворку, вимагаючи перезапуску всього фреймворку кожного разу, коли встановлюється нова служба. Крім того, якщо монітори потрібно запустити або зупинити під час виконання, це не можна зробити безпосередньо через AspectJ без перезапуску служби — те, що є небажаним у службах 24/7 .

У цю третю категорію, де монітор не міститься в коді, ми включаємо будь-який підхід до аналізу трасування, такий як система LogOs для моніторингу систем на основі OSGi, систем виявлення на основі журналу подій і систем журналювання. Головною перевагою цих підходів є слабкий зв’язок між властивістю та системою, що контролюється. Отже, якщо пакет замінено, монітор може спостерігати за ним у журналах, а відстежувані властивості залишаться незмінними для всієї системи. Крім того, опис майна розміщено в одному місці , що полегшує управління майном.

Однак такі системи агностичного кодування можна обійти, наприклад, - системи виявлення вторгнень і системи реєстрації можуть спостерігати лише за тим, які служби погоджуються надсилати. Якщо пакет надає послугу без запису достатньої кількості журналів, тоді монітор не має достатньо інформації для перевірки кореляції подій. Система LogOs краща, ніж обидві системи моніторингу, але ми побачимо, що деякі обмеження залишаються. Далі ми пояснимо кожну систему реєстрації.

1.5 Опис підходу до самовідновлювальних систем в D-SOA

Існуючий підхід до самовідновлюваної системи полягає у використанні структури D-SOA та розгляді заміни сервісу як механізму відкату [37]. При розгляді проблеми заміни послуг додаткова мета полягає в досягненні заміни без будь-якої модифікації коду клієнта. Класичні рішення в основному розглядають цю проблему і можуть бути згруповані в три основні категорії [39]: підхід на

основі абстракції, підхід на основі адаптера та гібридні рішення, що поєднують перші два. Ідея підходу, заснованого на абстракції, полягає у визначенні абстракцій вищого рівня, що означає конкретні служби, і доступ клієнтських програм до альтернативної конкретної служби замість доступу безпосередньо до наданої послуги. Навпаки, у підході на основі адаптера клієнтські програми отримують доступ безпосередньо до конкретної служби через адаптер. Нарешті, гібридний підхід може зменшити складність обслуговування, збільшивши кількість.

Який би підхід не використовувався, перша частина заміни зосереджена на пошуку нової послуги, яку можна використовувати замість недоступної. У автори пропонують алгоритм і дерева невідповідностей для пошуку несумісності на рівні інтерфейсів і протоколів відповідно. Перевірка сумісності між усіма доступними службами може зайняти багато часу та водночас вплинути на поточний бізнес-процес. Були запропоновані деякі підходи, що зменшують цю складність. Основна ідея полягає в тому, щоб зібрати доступні сервіси в групи сервісів, що пропонують однакові функціональні можливості , кожна клієнтська програма прив’язана не лише до однієї служби, а до групи послуг. У структурі SIROCO [46] використання семантичних анотацій мовою SA-WSDL для категоризації послуг в онтології OWL. У [39] автори пропонують групувати доступні служби для заміщення в групи, які називаються профілями. У [29] автори запропонували спільний інтерфейс (а саме Open Service Connectivity OSC) для збору веб-служб і динамічного зв’язування веб-служб. У цій роботі заміна веб-сервісів має два кроки: збирає функціонально подібні веб-сервіси в спільноти та змушує клієнтські програми підключатися до спільнот веб-сервісів за допомогою драйвера OSC.

Після пошуку потрібної послуги заміна тепер може бути реалізована, тобто друга частина. Здійснити заміну означає переналаштувати систему, щоб клієнтські служби могли продовжувати працювати за допомогою нової служби. У [17] автори пропонують алгоритм для реконфігурації служби CORBA, який передбачає пасивне посилання на недоступну службу та активне посилання на

нову службу, зберігаючи узгодженість програми та з деякими порушеннями виконання. У випадку послуг без громадянства це просто. Але для державних сервісів це складніше.

Оскільки постачальники послуг можуть робити будь-які припущення щодо наданих послуг з метою здійснення заміни служби без будь-яких наслідків з боку клієнта, посилання на служби можуть стати недійсними під час використання клієнтами після вивантаження служби. Ця проблема продуктивності спричинена динамічністю обслуговування систем D-SOA.

1.6 Опис платформи OSGi для дистанційного розгортання служб і сервісів

Платформа OSGi дозволяє дистанційно завантажувати та динамічно розгортати програми. Служба — це запущена реалізація Java, інтерфейс якої доступний у відкритому репозиторії та використовує посилання на службу замість самого об’єкта служби. Але це посилання також має недолік: службу, на яку посилається, можна зупинити, а її залежності застаріти на момент її використання, що призведе до застарілого посилання. Застаріле посилання — це посилання на послугу, яка більше не доступна через те, що пропозицію пакета цю послугу було зупинено, або пов’язану послугу було незареєстровано [9]. Клієнтські пакети можуть не знати про зникнення служби або посилання на служби застаріли під час виконання. Ми зосереджуємось на випадку мобільної платформи з OSGi, яка може виявити або втрачати з’єднання з деякими постачальниками послуг. У такому випадку запитана клієнтом послуга може бути втрачена під час використання.

Написання безпечного коду для обробки посилань на послуги OSGi зводиться до належного прослуховування реєстру служб OSGi та відстеження того, які служби є, а які – поза. Це також вимагає, щоб кожен виклик служби в коді клієнта виконував додаткові кроки. Це фактично викликає метод у службі, посилання на яку не застаріло. Це нелегко, як здається, оскільки задіяна

паралельність. Дійсно, потік може викликати службу, а інший скасовує її реєстрацію. Це легко руйнує захищений доступ до посилання на службу, якщо не використовуються внутрішні блокування або детальні повторні блокування читання/запису.

Щоб проілюструвати це, розглянемо клас, який є частиною основного API OSGI: org.osgi.util.tracker.ServiceTracker. Коротко кажучи, цей клас обробляє логіку відстеження появи та зникнення служби на основі набору інтерфейсів служби та фільтрів. Його можна використовувати для отримання одного або кількох посилань на службу в певний момент. Широко рекомендується використовувати його при роботі з рівнем обслуговування OSGi. Тим не менш, він не обробляє паралелізм, і багатопотокові пакети OSGi можуть використовувати застарілі посилання або створювати винятки, користуючись цим. Наступний фрагмент коду, частина демонстраційного активатора комплекту OSGi, створює виняток java.lang.NullPointerException, оскільки захищений доступ до посилання на службу є неправильним у цьому одночасному налаштуванні. Крім того, викликаючи два рази службу , ви можете отримати дві різні служби, які можуть генерувати помилки у випадку стану.

Наступний фрагмент коду такий самий, але отримане посилання на службу зберігається в пам’яті. Захищений доступ до посилання на службу може призупинитися під час обох викликів у цьому одночасному налаштуванні. Але це гарантує, що використана послуга в обох дзвінках є однаковою.

З цього блоку коду ми знаємо, що потік видавця також постійно публікує та видаляє службу, тоді як потік викликаючий постійно її викликає. Гонка умова призводить до того, що поточне використане посилання на службу стає застарілим. Метод hello() викликається вдруге через посилання незареєстрованої служби .

Це спостереження підкреслює той факт, що посилання на службу OSGi потрібно обробляти ретельно: легко зіткнутися з умовами змагання, коли кілька потоків виконуються одночасно, і легко виконати виклик методу за посиланням, який генерує NullPointerException за допомогою службовий трекер або застарів.

Перш за все, ми представили передумови цієї роботи та пояснили, чому ми -зосереджуємося на структурі OSGi. Однак, щоб контролювати комунікації використання служб без застарілих посилань у системах D-SOA, ми намагаємося використовувати класичні системи моніторингу для моніторингу. Ми перерахували деякі споріднені роботи класичних систем моніторингу в першій частині сучасного стану. Вони мають свої рекламні переваги та недоліки. Але їх недостатньо, щоб задовольнити динамічність послуг у системах D-SOA. Крім того, корисним підходом може бути відмовостійка технологія уможливити самовідновлення служб у динамічній системі на основі SOA. Ми -представили кілька класичних рішень для динамічної заміни служб із збереженням/без стану в системах на основі D-SOA без будь-яких наслідків для клієнта. Це робить сервіси більш автономними та динамічними. Але застарілі посилання та виключення нульового покажчика відбуватимуться під час виконання в системах D-SOA через динамічність служби, наприклад, у системах на основі OSGi. Оскільки клієнт не знає, чи були ці речі запущені чи ні, вони можуть призвести до неправильних результатів або навіть до збою системи. Існують деякі підходи, які намагаються впоратися з динамічністю в OSGi. Але це не повне вирішення цих проблем.

Сервісно-орієнтована архітектура — це підхід, за якого програмні системи розробляються з точки зору складу сервісів. OSGi — це сервіс-орієнтована структура, призначена для цілодобових систем Java. У цьому сервісно-орієнтованому підході програмування програмне забезпечення складається зі служб, які можуть динамічно з’являтися або зникати. У такому випадку не можна використовувати класичні підходи моніторингу зі статичним впровадженням моніторів у служби. У цьому розділі ми пропонуємо підхід динамічного моніторингу, присвячений локальним системам SOA, зосереджуючись, зокрема, на OSGi. По-перше, ми визначимо дві ключові властивості слабозв’язаних систем моніторингу: стійкість до динамічності та комплексність . Далі ми пропонуємо інструмент OSGiLarva, який є реалізацією, націленою на структуру OSGi. Нарешті, ми представляємо деякі кількісні результати, які показують, що

динамічний монітор на основі динамічних проксі-серверів і інший на основі аспектно-орієнтованого програмування мають еквівалентні характеристики.

Висновки до розділу

В даному розділі представлено підходи до реалізації SOA та порівнює їх між собою. Також наведені сучасні відомості про різноманітні монітори для перевірки безпеки використання послуг статичних або динамічних систем. Представлено класичні підходи, намагаючись виконати динамічні заміни сервісу за допомогою сервісів із збереженням або без стану на стороні сервера.

РОЗДІЛ 2. Проектування структура моніторингу архітектур для підтримки динамічності послуг використання паттернів проектування

2.1 Основні відомості моніторингу SOA систем

Відомо, що служби слабко пов’язані, і клієнт викликає методи служби, якщо цей інтерфейс служби відповідає системі на основі SOA. Моніторинг критичної системи на основі D-SOA є складним завданням. Існує багато інструментів моніторингу виконання, але їхні властивості вводяться в систему під час кодування або під час завантаження (монітор виконання [18], Larva [22] і JavaMOP [25]). Це означає, що коли відстежувана служба на основі системи D-SOA зникає або замінюється під час виконання після зв’язування, її відстежувана властивість також видаляється із системи, якщо відстежувана система не перезавантажується. Крім того, ці роботи не розглядають вираження властивостей у термінах рамкових подій.

В цьому розділі ми пропонуємо застосувати динамічний підхід до систем моніторингу виконання шляхом вставки моніторів у точці зв’язування клієнт-сервер, а не «статично» під час компіляції чи завантаження. Це означає, що як прив’язки до послуг, так і прив’язки для моніторингу поведінки є динамічними та слабко пов’язаними, таким чином підтримуючи заміну послуг. Цей підхід дозволить зберегти стани моніторингу поведінки в різних версіях служби та перевірити, чи обидві версії поведінково сумісні.

Інша серйозна проблема у дуже динамічному контексті, де реалізація інтерфейсу може бути замінена, полягає в тому, щоб жодна реалізація або її частина не могла обійти структуру моніторингу. Зауважте, що якби це сталося, монітор не зможе виявити шкідливий код, який може бути виконано. Крім того, який висновок можна зробити про спостереження системи, якщо деякі події

могли бути пропущені? Наша мета полягає в тому, щоб система моніторингу була повністю активною, навіть якщо постачальник послуг ігнорує це.

У цьому контексті ми припускаємо, що динамічний монітор часу виконання повинен мати дві важливі риси: стійкість до динамічності та комплексність. Зауважте, що ми не припускаємо, що кожна служба поводиться належним чином, а лише те, що якщо авторизовану службу потрібно перевірити на певну властивість, тоді жодна подія поведінки служби не може обійти спостереження монітора. З цієї причини архітектура покладається на загальний механізм перехоплення подій і динамічний, слабко пов’язаний механізм підключення для перевірки автоматів .

Внесок цього розділу – це загальний підхід, а також інструмент, заснований на OSGi. У цьому інструменті логіка верифікації автомата обробляється за допомогою адаптації існуючого інструменту моніторингу Larva [32]. Нарешті, введення динамічності в монітор також збільшує обсяг властивостей, які ми можемо розглянути. Таким чином, ми вводимо деякі динамічні примітиви в мову опису властивостей, щоб зробити можливим опис поведінкових властивостей, де реєстрація/скасування реєстрації служби є подіями, які можна виразити. Крім того, ми також змінюємо життєвий цикл властивостей, оскільки за різних обставин стан монітора може знадобитися зберегти або скинути під час заміни основної служби.

Щоб полегшити розуміння нашого внеску, ми наведемо приклад системи з динамічним моніторингом, яка відповідає нашій пропозиції. Розглянемо вбудований клієнт на мобільному пристрої на базі динамічної платформи SOA, якому необхідно спілкуватися з віддаленою системою за певним протоколом (рис. 2.1). Нехай дві служби S і та S2 забезпечують ідентичний інтерфейс для доступу до віддаленої системи через різні носії: S і за допомогою з’єднання WiFi, а S2 за допомогою з’єднання 3G. З такою конфігурацією ми можемо вважати, що кожного разу , коли Wi-Fi-з’єднання вимикається, система скасовує реєстрацію S і , фактично перемикаючи клієнта на S2 і навпаки.

Крім того, ми вважаємо, що використання дистанційної системи вимагає клієнта автентифікується за допомогою служби, і деякі системні дії мають виконуватися -атомарно.

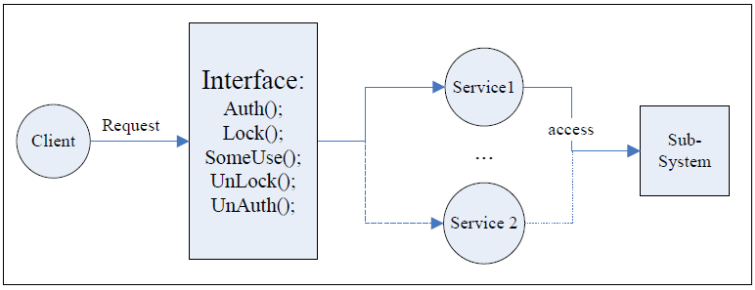


Рисунок 2.1 - Динамічна система SOA, що підтримує заміну послуг

У такому прикладі можливість заміни послуги є вирішальною. Потім ми пропонуємо на рис. 2.2 приклад сценарію виконання, який має підтримуватися системою. У цьому сценарії служба S і замінюється на S 2 під час атомарної частини циклу.

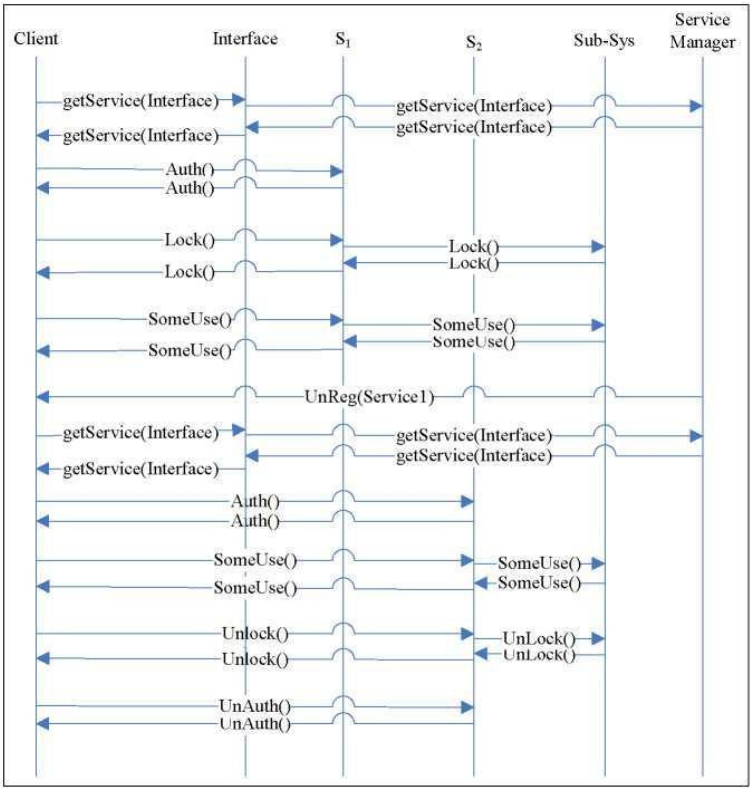
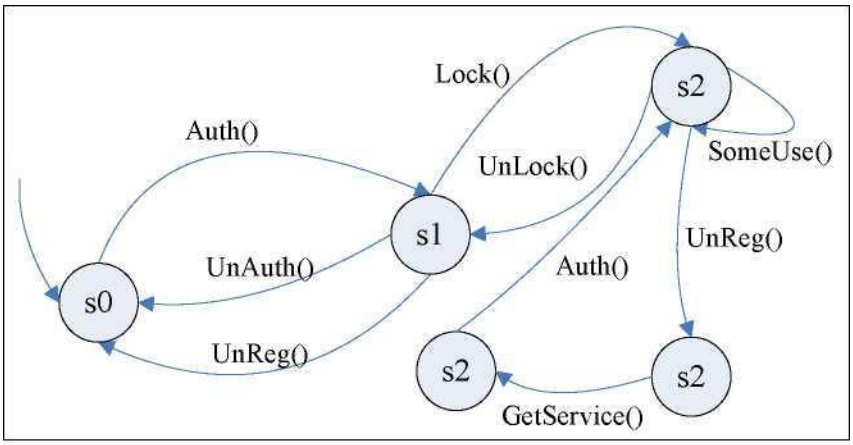
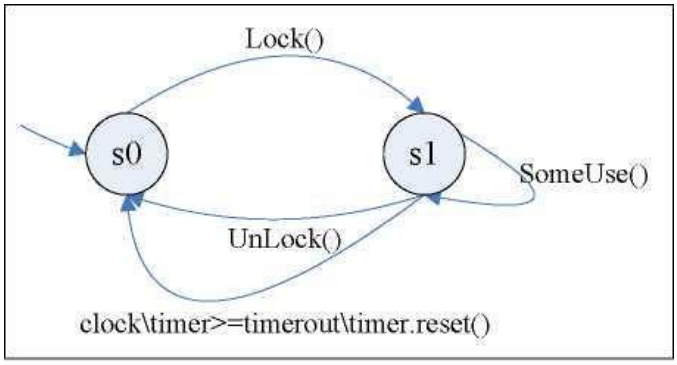


Рисунок 2.2 - Приклад сценарію з динамічно відстежуваною системою, що підтримується прикладом на рис. 2.1

В іншій частині ми можемо описати правильне використання системи в деякій властивості та перевірити це шляхом моніторингу під час виконання. Наприклад, дві наступні властивості виражають очікувану поведінку, описану раніше: (i) клієнт проходить локальну автентифікацію в службі перед її використанням, і (ii) конкретне використання підсистеми вимагає, щоб клієнт відкрив замок і закриває його після використання. У цьому прикладі хочеться переконатися, що виконання, описане на рис. 2.2 , є правильним щодо цих властивостей.



A. Client-side: instance property



B. Interface-side: class property

Рисунок 2.3 - Приклад властивості, пов’язаної з прикладом на рис. 2.1

Ці властивості можна описати парою автоматів (рис. 2.3), але з різною інтерпретацією кожного. Автомат локальної автентифікації (рис. 2.3 A) підтримується у разі заміни служби та має бути створений для кожного окремого

клієнта, який використовує систему. Далі ми будемо називати такі властивості властивостями екземпляра, оскільки вони створюються на основі кожного об’єкта; в даному випадку клієнт. Навпаки, управління атомарним використанням підсистеми (рис. 2.3.B) має бути централізованим і спільним для всіх клієнтів. Навіть якщо службу буде видалено та замінено, ми хочемо зберегти поточний стан підсистеми в пам’яті. Далі ми називаємо такі властивості властивостями класу , оскільки їх час життя охоплює весь життєвий цикл системи і не прив’язаний до конкретної сутності.

Підводячи підсумок, наша пропозиція полягає в тому, щоб надати структуру моніторингу, яка здатна контролювати такі властивості, прослуховуючи виклики методів і події інфраструктури OSGi динамічним, стійким і комплексним способом.

У першій частині цього розділу ми описуємо абстрактну архітектуру моделі системи моніторингу, що підтримує специфічні функції динамічних систем SOA, і обговорюємо її характеристики. У другій частині розглядаються динамічні примітиви з динамічної системи SOA. Наприкінці ми надаємо загальний опис властивостей нашої моделі монітора для контрольованої системи з трьох точок зору: сторона сервера, сторона клієнта, сторона інтерфейсу служби.

2.2 Опис підходу пропонованої загальної архітектури на основі паттернів

Наша пропозиція полягає в тому, щоб динамічно вставляти проксі-сервер моніторингу перед кожною службою та виконувати монітори в деяких автономних службах (рис. 2.4). Коли відбувається подія використання служби, сповіщення надсилається кожному пов’язаному монітору, який перевіряє подію на відповідність властивостям.

Цікавою перевагою використання динамічного проксі над AspectJ є те, що ми можемо почати або зупинити моніторинг властивості без перезапуску служби. Дійсно, оскільки проксі-сервер прив’язується до запиту служби, це можна легко

впоратися, тоді як аспекти AspectJ прив'язуються принаймні під час завантаження класу, вимагаючи перезапуску служби.

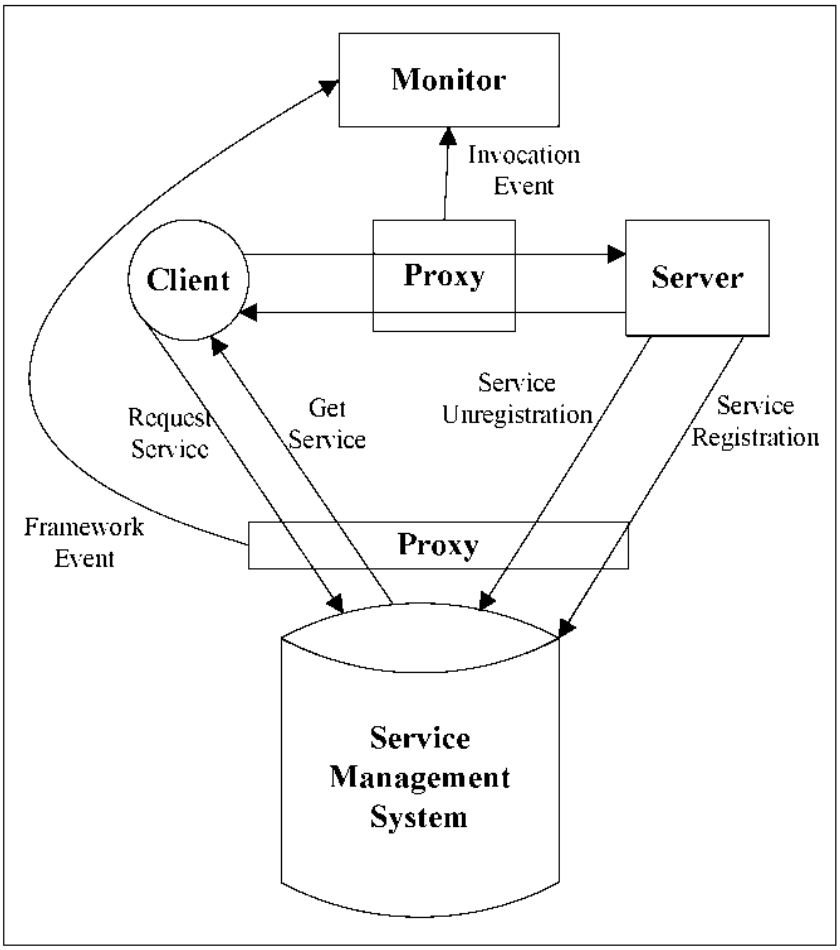


Рисунок 2.4 - Запропонована абстрактна архітектура системи моніторингу

Оскільки служби розглядаються як чорні скриньки з точки зору робочого середовища, така архітектура розроблена таким чином, щоб враховувати лише властивості їх зовнішнього інтерфейсу. Це відповідає властивостям, що виражають нормальне авторизоване використання послуги. Однак, оскільки ми розглядаємо динамічні системи, ми також хочемо розглянути спеціальні рамкові події, такі як скасування реєстрації послуги або отримання нової послуги. У цій програмі roach ми зосередимося на поведінкових властивостях.

Оскільки кілька клієнтів можуть працювати одночасно в рамках інфраструктури, область властивостей не повинна обмежуватися використанням одного клієнта. Розглядаємо можливість додавання монітора перед декількома

клієнтами. Розглядаючи як моніторинг властивостей екземпляра, так і властивостей класу, ми надаємо можливість одночасно перевіряти як локальні, так і глобальні властивості в системі.

Щоб увімкнути властивості, виражені в термінах подій виклику методу та -подій фрейму (запити, реєстрація, скасування реєстрації тощо), нам потрібно зафіксувати обидва види подій — події між клієнтом і службою та деякі події з система реєстрації послуг. Щоб встановити монітор між службою та клієнтом, який її використовує, ми адаптуємо структуру, щоб зробити це невидимим як для клієнта, так і для послуги. Двома цікавими характеристиками цього підходу є те, що він не змінюємоніторингу для підтримки динамічності послуг.

Двійковий підпис служби та що ні служба, ні клієнт не знають про потенційно запущений монітор. Додавши ще один проксі-сервер перед системою керування послугами фреймворку, ми отримуємо сповіщення про запити на отримання довідок про послуги.

Далі ми глибше розглянемо наші два основні принципи.

Стійкість до динамічності. Оскільки система моніторингу знаходиться в автономній службі, монітори відокремлені від коду. Коли в структурі відбуваються зміни, механізм спостереження та його властивості залишаються незмінними.

Комплексний моніторинг. Одна з головних концепцій динамічної SOA полягає в тому, щоб мати структуру, яка дозволяє динамічно завантажувати та вивантажувати слабозв’язані служби. Оскільки фреймворк відповідає за реалізацію кожного запиту на службу, фреймворк може додати проксі-сервер між клієнтом і службою для спостереження за їхнім спілкуванням. Це спостереження є вичерпним, і жодне спілкування не може оминути цей проксі-сервер.

Монітор запускається, коли відстежувана служба реєструється в рамках. З цього моменту кожна подія, пов’язана з цією службою (наприклад, виклик методу служби, завантаження служби тощо), передається на цей монітор. Оскільки ми знаходимося в динамічній системі, можуть відбуватися динамічні

події, наприклад, скасування реєстрації зареєстрованої служби або завантаження зареєстрованої служби. Ми пропонуємо ввести наступні чотири примітиви:

- Реєстрація : ця подія відбувається, коли нова реалізація служби реєструється на фреймворку. Це означає, що тепер клієнт може отримати цю послугу в будь-який час. Якщо інша реалізація вже зареєстрована, вона має ту саму властивість інтерфейсу.
- Get S service : ця подія відбувається, коли клієнт запитує послугу. Це може призвести до двох ситуацій: клієнт отримує послугу або клієнт не отримує жодної послуги. Якщо клієнт не зміг отримати послугу з сервера, це означає, що немає зареєстрованої служби, яка відповідає запиту клієнта. Ми представляємо подію СЛУЖБИ N O G ET S для обробки цього випадку.
- UNGET S SERVICE: ця подія відбувається, коли клієнт випускає завантажену службу. Кожен клієнт може випустити свій об’єкт служби відповідно, і ця служба також існує в пам’яті для завантаження та використання іншими клієнтами.
- Скасувати РЕЄСТРАЦІЮ : ця подія відбувається, коли послугу скасовано з реєстрації . Створений службовий об’єкт тоді вважається знищеним. Однак, якщо клієнти все ще користуються цією послугою, ці дії вважаються такими, що, можливо, більше не безпечні чи нефункціональні.

2.3 Загальний опис властивостей архітектрури

У цій частині обговорюється мова опису властивостей і зосереджується на масштабі опису властивості, головним чином викликаному розташуванням пов’язаного з ним монітора. Дійсно, оскільки ми не в системі з одним клієнтом і однією службою, у нас може бути багато клієнтів, які використовують багато послуг одночасно. У такому випадку місце розташування монітор може змінити точку зору властивості і, отже, його виразність. Як загальну точку зору для опису властивостей, її можна визначити принаймні трьома можливостями (наприклад, рис. 2.5):

1) точка зору клієнта,

2) точка зору реалізації сервісу,

3) точка зору інтерфейсу.

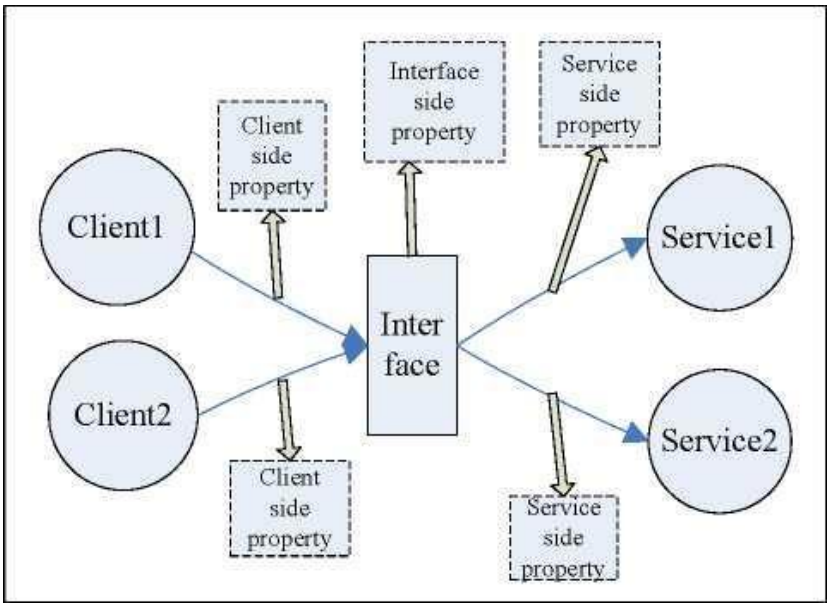


Рисунок 2.5 - Можлива точка зору для властивостей

Далі ми надамо деталі опису властивості з трьох точок зору моніторингової системи.

Якщо розробник описує властивість з цієї точки зору, як показано на рис. 2.6, він/вона розглядає використання однієї служби. Легко розглянути певну поведінкову залежність у деяких паралельних використаннях кількома клієнтами. Однак, оскільки ми розглядаємо властивості на основі автоматів, неочевидно, як розрізнити клієнтів у властивості. Крім того, складно розглядати використання кількох реалізацій інтерфейсу одночасно, з потенційно деяким зв’язком між ними.

Що стосується динамічної частини, то не інтуїтивно зрозуміло описувати та використовувати той факт, що на платформу завантажено нову реалізацію того самого інтерфейсу служби. Більше того, здається складним ділитися пам’яттю властивостей між реалізаціями того самого інтерфейсу. Отже, якщо послугу замінено, немає засобів збереження її властивості в пам’яті з її внутрішнім станом

і відображення її на іншій реалізації, призначеній для продовження розпочатої роботи.

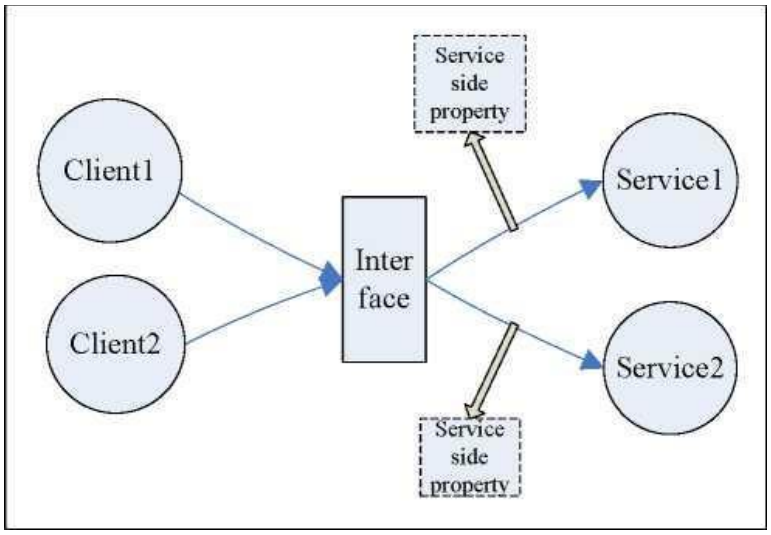


Рисунок 2.6 - Опис властивості: точка зору реалізації сервісу

- Простота опису поведінки кожної реалізації служби без необхідності встановлення зв’язку з іншими можливими реалізаціями.
- У випадку служб із збереженням стану, з різними адресними просторами пам’яті для кожної реалізації , дуже легко описати систему.

Недоліки:

- Складність опису спільної пам’яті між службами.
- Неможливість описати загальну поведінку для кожного клієнта, оскільки ми не можемо розрізняти клієнтів.

2.4 Дослідження властивостей з точки зору сервісного інтерфейсу та користувача

З цієї точки зору ми розглядаємо те, що можна зробити через сервісний інтерфейс, показаний на рис. 2.7 . Легко описати глобальне використання будь-якої реалізації цього інтерфейсу будь-яким клієнтом, але не робити відмінностей між клієнтами або між використовуваними реалізаціями.

За своєю природою така властивість не пов’язана безпосередньо зі службою і, таким чином, описує властивість, спільну для всіх реалізацій. Зауважте, що легко розглянути завантаження або вивантаження реалізації служби, навіть якщо це заміна, яка бажає зберегти поточний стан властивості.

Оскільки наша мова опису властивостей базується на автоматах, єдиний спосіб розглядати паралельне використання багатьох клієнтів — створити певну композицію між властивістю та собою. Однак така техніка призводить до комбінаторного вибуху розмір автомата. Крім того, це обмежує максимальну кількість клієнтів і послуг, оскільки нам потрібна ця інформація, щоб створити композицію.

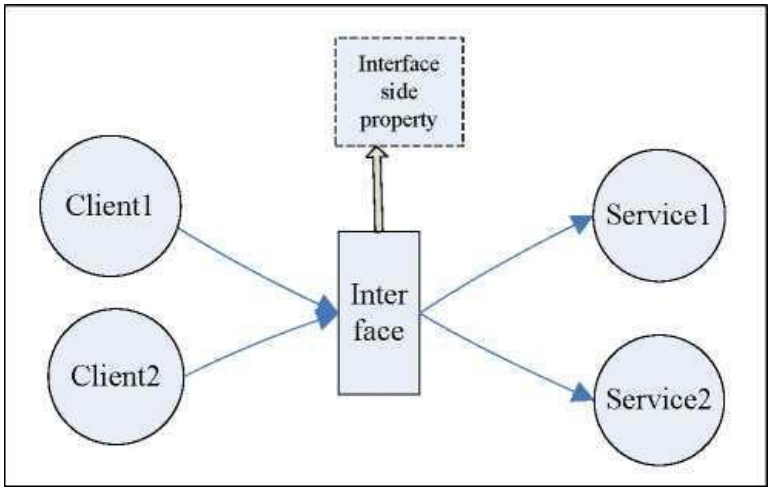


Рисунок 2.7 - Опис властивості: точка зору інтерфейсу служби

Переваги:

- Легко зробити опис дозволених видів використання з глобальної точки зору
- Легко розглядати завантаження/вивантаження реалізацій
- Можливість спільного використання одного стану властивості між реалізаціями служби. Недоліки:
- Ризик збільшення розміру опису спільної властивості, якщо ми хочемо описати одночасну поведінку кількох клієнтів.
- Неможливість описати загальну поведінку для кожного клієнта, оскільки ми не можемо розрізняти клієнтів

Ця третя можливість передбачає, що кожен клієнт має власний екземпляр властивості (рис. 2.8). Отже, легко описати правильне використання служби з точки зору одного клієнта та розглянути скільки завгодно паралельних використань без будь-якого комбінаторного вибуху.

Крім того, легко описати використання кількох послуг одним клієнтом і поведінкову залежність у разі одночасного використання послуг. У разі підміни послуги такий підхід може бути стійким, оскільки майно закріплюється за клієнтом.

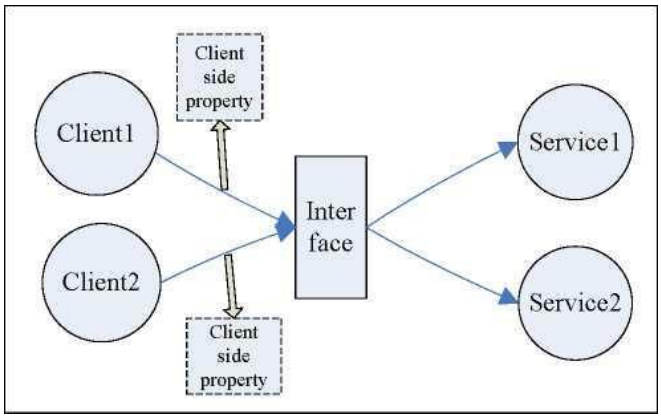


Рисунок 2.8 - Опис властивості: точка зору клієнта

Однак у разі одночасного використання однієї послуги декількома клієнтами, якщо між цими використаннями існує певна взаємодія, описати її складніше.

Переваги:

- Легко зробити опис авторизованого використання конкретного клієнта
- Легко розглядати завантаження/вивантаження реалізацій
- Можливість спільно використовувати один стан властивості між декількома реалізаціями служби
- Немає ризику збільшення розміру спільного майна, оскільки його неможливо описати

Недоліки:

- Складність опису глобальної поведінки, включаючи кілька клієнтів

Отже, ці три точки зору доповнюють один одного. Якщо кожен із них окремо використовується для опису властивостей динамічних систем SOA, цього недостатньо.

2.5 Реалізація моделі системи моніторингу в контексті фреймворку OSGi

Ми пропонуємо конкретну реалізацію описаної моделі системи моніторингу в контексті фреймворку OSGi: OSGiLarva (рис. 2.9). У нашому інструменті ми використовуємо механізм Java

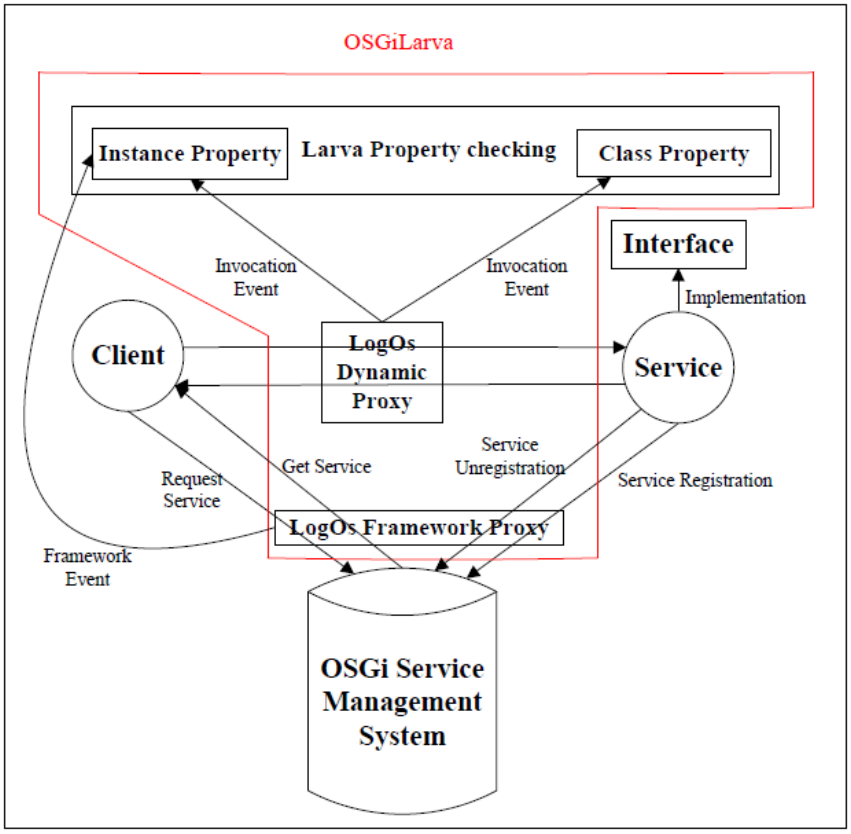


Рисунок 2.9 - Реалізація OSGiLarva

Механізми, щоб створити проксі-сервер між кожним клієнтом і службою. Цей проксі-сервер динамічно генерується з проксі-сервера фреймворку, підключеного до фреймворку OSGi, і прослуховує всі події фреймворку, такі як впровадження нової служби або запит на послугу клієнтом.

Ця реалізація об’єднує два існуючі інструменти: Larva [32] і LogOs. Інструмент Larva належить до м’якого кодування, а система LogOs – до агностичного кодування. Ми використовуємо LogOs як механізм підключення для спостереження за взаємодією служб. Larva — це компілятор, який генерує систему перевірки, виражену мовою Java. Ми будемо використовувати адаптацію Larva для перевірки подій властивостей, які передаються LogOs.

Ми описуємо реалізацію монітора з наступних частин: спочатку ми представляємо виразність властивостей за допомогою динамічних примітивів системи OSGiLarva, а потім пояснюємо мову опису властивостей OSGiLarva. Далі ми представляємо реалізацію OSGi Larva як з LogO, так і з системою Larva . Нарешті, ми описуємо, як процес реєстрації послуги в OSGi враховуватиме існуючу монітор властивостей, щоб вставити його між споживачем послуги та самою службою.

Мова опису OSGiLarva спочатку базується на мові опису властивості Larva. Ми адаптували його, щоб підтримувати більше динамічності. Ця адаптація - виконується за допомогою трьох розширень. Перший — це введення примітивів каркасних подій і властивості як композиції властивостей класу та властивостей екземпляра в мові опису властивостей. Другий виражає синтаксис і семантику автоматів OSGiLarva. Останній описує повну властивість OSGiLarva мовою опису властивостей OSGiLarva.

Larva використовує як вхідні дані мову опису властивостей, засновану на автоматах, розширену таймерами, змінними та діями. У самій властивості користувач визначає набір символів, які використовуються в автоматах. Ці символи є подіями, які в оригінальній версії Larva визначені в термінах імен методів. Таким чином, ми пропонуємо використовувати динамічні примітиви, описані в у визначенні подій, щоб увімкнути рамкову подію: Register, UnRegister, GetService, NoGetService. UngetService не входить до них, оскільки вимагає, щоб клієнт використовував деякі інтерфейси OSGi framework для захоплення операції UngetService. Використання UngetService та GetService інтерфейсу ServiceFactory є частиною нашої майбутньої роботи. Наразі ми зосереджуємося лише на інших

подіях, які відповідають описам подій, створеним адаптованою версією LogOs. Отже, LogOs потрібно зареєструвати деяких слухачів у фреймворку .

Перед перевагами та недоліками підходів до виразних властивостей, ми пропонуємо розглядати властивості як комбінацію двох типів властивостей для OSGiLarva, пов’язаних з двома точками зору: стороною клієнта та стороною інтерфейсу. Ці дві точки зору в нашому моніторі відповідно називаються властивістю екземпляра та властивістю класу. Ми пропонуємо не розглядати точку зору служби, оскільки при типовому використанні OSGi, якщо кілька служб реалізують один інтерфейс, фреймворк надає перевагу використанню однієї реалізації всіма клієнтами. Крім того, з нашого досвіду ми припускаємо, що властивості зазвичай є стороною клієнта, оскільки властивість інтерфейсу не може розглядати одночасне використання служб багатьма клієнтами без вибуху стану. Нарешті, щоб мати можливість додати централізовану властивість, властивості інтерфейсу можуть бути корисними для вираження деяких спільних обмежень, таких як системи блокування/розблокування.

Оскільки наш внесок базується на мові опису Larva [31], обраній через її близькість до наших вимог, ми головним чином орієнтуємо нашу пропозицію відповідно до Larva та адаптуємо її для підтримки більшої динамічності. У Larva властивості описуються автоматами, де один файл сценарію може містити кілька автоматів. Крім того, Larva надає у своїй мові можливість визначення параметризованих автоматів, які можуть бути створені за допомогою параметрів подій за допомогою ключового слова FOREACH. Ми використовуємо ці характеристики, щоб використовувати властивості, що складаються з двох частин (властивість екземпляра та властивість класу):

- Властивість екземпляра: якщо властивість визначено як властивість екземпляра, тоді щоразу, коли новий клієнт отримує доступ до інтерфейсу, новий екземпляр властивості генерується та додається в монітор. Коли клієнт завершує роботу, пов’язаний екземпляр властивості також можна видалити. Отже, хоча такі властивості все ще стійкі до динамічності реалізацій сервісу, вони навмисно

не стійкі до динамічності клієнтів. Структурні події корисні для опису фактичного стану та поведінки кожного клієнта.

• Властивість класу: цей випадок відповідає централізованій властивості, тобто кілька клієнтів, які використовують певний інтерфейс, спільно використовуватимуть одну властивість класу. Така властивість більш стійка до динамічності, оскільки властивість класу можна зберегти у пам’яті, доки пов’язаний інтерфейс не буде вивантажено. Як такий, він не пов’язаний із взаємодією конкретного користувача чи конкретною реалізацією служби, і тому може використовуватися, наприклад, для вираження деяких централізованих механізмів блокування/розблокування. Необхідно описати виклики методів у Class-Property. Події рамки, які будуть описані у властивості екземпляра для кожного клієнта, до якого здійснюється доступ, марні у властивості класу. Однак, якщо кілька реалізацій використовуються одночасно, їх, ймовірно, потрібно буде синхронізувати.

Далі ми представляємо нашу адаптовану структуру мови опису властивостей Larva в контексті OSGiLarva для більшої динамічності.

У цьому розділі ми пропонуємо формально визначити властивості OSGiLarva в термінах правильного та поганого відстеження виконання. Трасування виконання завершується помилкою тоді і тільки тоді, коли воно призводить до того, що автомат властивості досягає поганого стану (визначеного у властивості). Трасування виконання є правильним, якщо всі досягнуті стани автомата є лише непоганими станами.

Властивості екземпляра та властивості класу схожі за визначенням. Вони відрізняються лише життєвим циклом. Потім ми спочатку визначаємо, що таке властивість з її синтаксисом і семантикою, перш ніж збільшити їх особливості.

2.6 Мова опису властивостей OSGiLarva

У Larva одним із способів ввести новий контекст є використання пропозиції FOREACH . Це речення є кількісним визначенням об’єкта. Отже, для кожного екземпляра даного класу Larva генерує новий екземпляр внутрішньої

властивості. Крім того, для того, щоб розрізнити користувачів, класичній Larva потрібна інформація, явно надана абонентом, наприклад ідентифікатор сеансу, який передається як параметр. Отже, Larva має лише ту саму інформацію, що й реалізація служби для перевірки властивості.

Ми пропонуємо адаптувати структуру FOREACH для наших потреб, ввівши новий пункт: foreachclient. Ця конструкція базується на адресі абонента та генерує новий контекст для кожного абонента. Як приклад, таке положення може дозволити перевірити відсутність підробки сесії ID, що неможливо в larva. Описану властивість у контексті FOREACHCLIENT буде повторно створено для кожного клієнта за допомогою відстежуваної служби. Це частина властивості.

Дуже важлива відмінність між пропозиціями FOREACH і FOREACHCLIENT полягає в тому, що перше базується на значеннях, обчислених всередині пропозиції ПОДІЙ із спостережуваних параметрів, тоді як друге базується на значеннях, обчислених і наданих LogOs, відповідно до його спостережень.

Ключове слово FOREACHCLIENT приймає два параметри: Long pid і String itfName. pid — це числовий ідентифікатор клієнта, пов’язаного з поточним екземпляром властивості. Конкретно, ми використовуємо ідентифікатор процесу як ідентифікацію. itfName — це ім'я інтерфейсу служби, пов'язаного з поточною подією. Значення обох параметрів безпосередньо передаються зі спостереження LogOs. Оскільки FOREACHCLIENT є розширенням пропозиції FOREACH, ми зберігаємо всі мовні характеристики останнього.

Окрім речення foreachclient, нам потрібно пояснити ГЛОБАЛЬНЕ речення. Властивість кожного інтерфейсу служби створюється лише один раз, а потім використовується всіма клієнтами. Ці властивості є класами-властивостями. Ці властивості будуть виражені в реченні GLOBAL .

Можна виразити властивість у термінах кількох сервісних інтерфейсів. Щоб це зробити, достатньо написати стільки властивостей у глобальному пункті, скільки сервісних інтерфейсів для перевірки. Зв’язок між відстежуваними інтерфейсами та властивостями синтаксично здійснюється за допомогою назви

властивостей, яка має бути назвою інтерфейсу. Ці властивості поділяють визначення змінної та визначення подій у глобальному контексті. Він може спілкуватися з усіма «примірниками» властивості.

Для того, щоб розрізнити однакові імена методів, які надаються різними інтерфейсами, ми розробили формат дійсного визначення події в кожному пункті EVENTS , щоб виразити точний виклик методу. Наприклад, eventName = {interfaceName- methodName(type, ...)}, де eventName — визначена назва події, interfaceName- methodName(type,...) — це виклик методу, а type list — це типи параметрів методу.

У наступному розділі ми виражаємо опис властивості OSGiLarva та відстежуємо задані траси виконання для прикладу системи.

2.7. Приклад застосування з використанням фреймворку OSGiLarva

У цьому розділі ми надаємо приклад системи бронювання авіакомпаній із її властивостями та пропонуємо відстежувати правильні чи ні правильні сліди виконання. Цей приклад (рис. 2.10) складається з двох сервісних інтерфейсів: бронювання та оплати. Інтерфейс резервування забезпечує два методи selectFly(String) і confirm(boolean) і реалізується двома службами S1 і S2 . Платіжний інтерфейс має єдину реалізацію служби S3 , що забезпечує два методи pay (String) і resetPay().

Клієнти повинні використовувати обидва методи послуг із спеціальним замовленням, наприклад, перед оплатою клієнти повинні вибрати флай або перед підтвердженням клієнти повинні оплатити. Якщо поточна послуга бронювання (тобто S1) скасовується до підтвердження бронювання авіакомпанії клієнта, платіжну послугу потрібно налаштувати повторно, а потім клієнт отримує нову послугу бронювання (тобто S2) із початкового стану щоб забронювати авіаквиток.

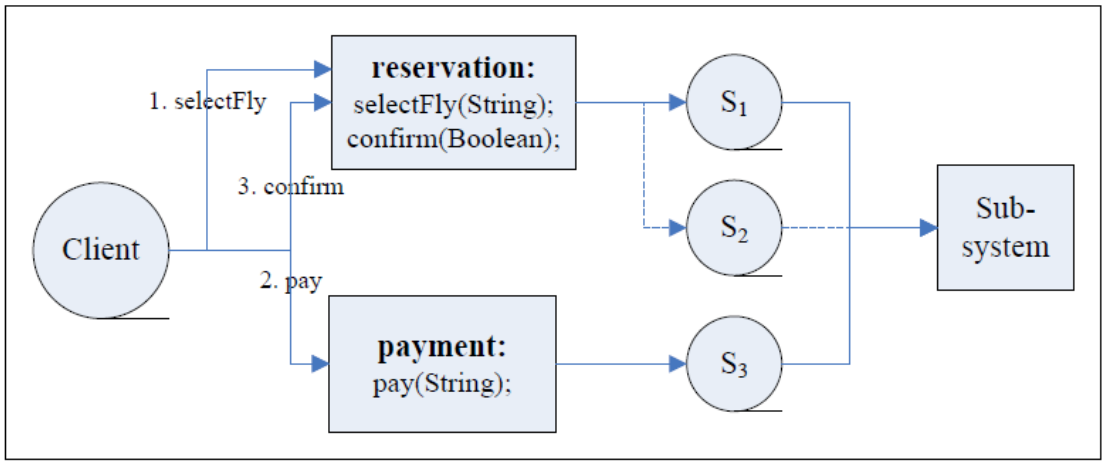


Рисунок 2.10 - Моніторинг використання послуг

Спочатку ми наведемо частину файлу властивостей (показаного на рис. 2.11), який складається з властивості екземпляра (тобто властивості клієнта) і двох властивостей класу (тобто одна властивість на інтерфейс служби) відповідно до цього прикладу.

Потім ми даємо визначення автомата властивості OSGiLarva властивості clients , яка є властивістю екземпляра на рис. 2.11. Згідно з визначенням, перехід визначається у вигляді: £(s, e) = [(c, a, s')], де s - початковий стан, e - подія, s' - досягнутий стан , c — умова, a — дія. Оскільки цей автомат властивостей описує властивість частини екземпляра для кожного клієнта, ця властивість описана в реченні FOREACHCLIENT . Змінна itfName є параметром foreachclient. У цьому контексті автомат властивостей цього прикладу можна визначити так:

У цьому прикладі автомата OSGiLarva «true» означає, що цей перехід не має умов, «nop» означає відсутність жодних дій у цьому переході. Графічне зображення цього автомата властивостей показано на рис. 2.11.

Щоб продемонструвати повне використання системи OSGiLarva, ми пропонуємо спостерігати дві траси виконання до та ti. Вони вибираються так, що один правильний, а другий закінчується помилкою. Оскільки значення контекстної змінної "itfName" надається LogOs автоматично, ми додаємо його як нижній індекс подій введення, коли це необхідно.

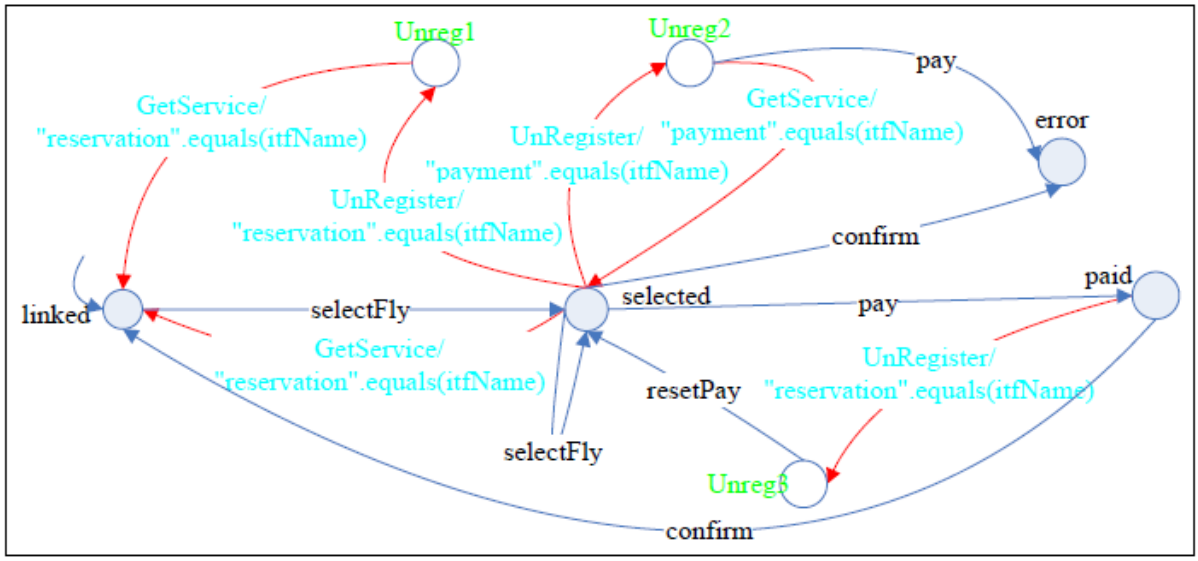


Рис. 2.11. Автомат OSGiLarva на стороні клієнта для бронювання авіакомпаній

$$t_1=[reservation.selectFly(String), UnRegister_{itfName="payment"}, payment.pay(String)]$$

Щоб використовувати OSGiLarva для перевірки цих трасувань, ми спочатку маємо перекласти ці траси як траси подій відповідно до пункту властивості властивості, надаючи наступні траси подій t0 і ti, які можна відтворити у властивості автомата.

Оскільки реалізація нашого монітора базується як на системах LogOs, так і на системах Larva, ми спочатку надамо більше корисних деталей про обидві системи. Потім ми представляємо нашу адаптацію LogOs для перехоплення взаємодії сервісів і надаємо деякі подробиці про наші модифікації Larva.

Структура LogOs У системі LogOs є чотири складові частини: анотації , перехоплення, журналювання, зберігання. Кожна частина має свої обов'язки. Частина анотацій спрямована на визначення доменно-залежної мови (DSL) за допомогою чотирьох анотацій Java, які змінюють стандартну поведінку протоколювання викликів методу інтерфейсу служби. Перехоплювальна частина відповідає за перехоплення викликів під час роботи системи. Додавання визначених анотацій в інтерфейс сервісу. Коли вказаний метод служби або

параметри методу позначені на стороні інтерфейсу, усі вони будуть перехоплені під час роботи.

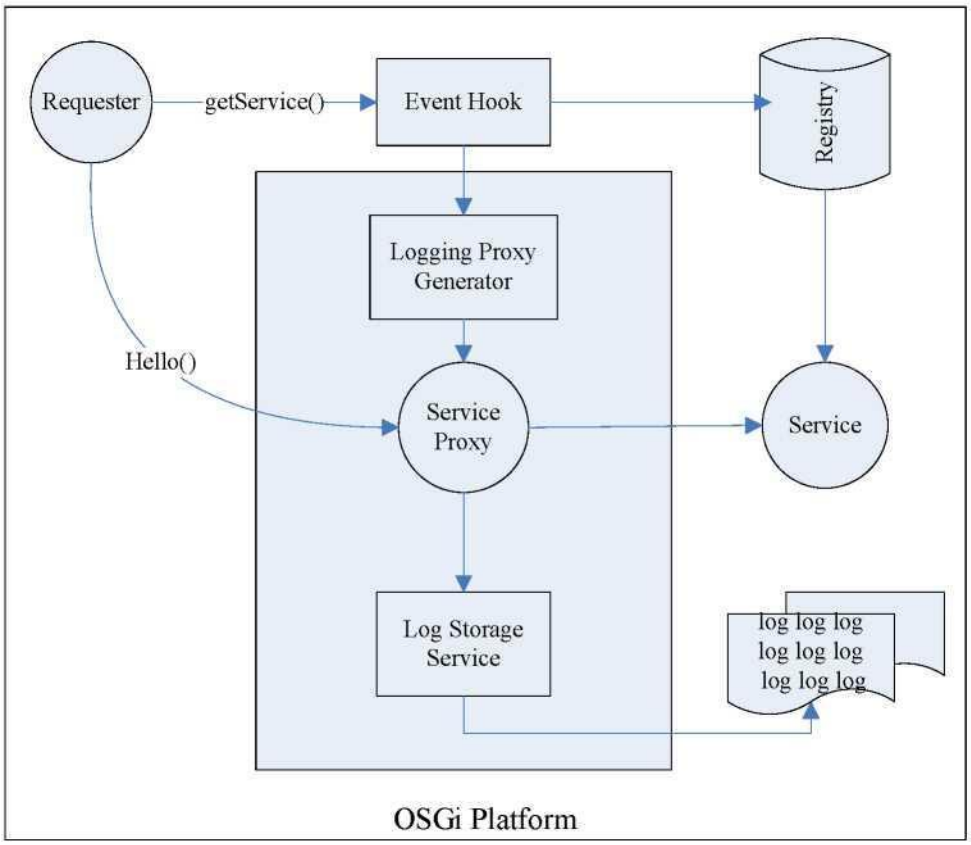


Рисунок 2.12 - Обробка системи LogOs працює для системи на базі OSGi framework

Частина реєстрації призначена для відстеження перехоплених дій або параметрів і запису цих деталей інформації. Він включає в себе: унікальний ідентифікатор, час, назву методу і так далі. Частина зберігання відповідає за виведення відстежуваного запису для користувачів. Файл запису має назву logosng-recorded-on-1368691248161.cur. Це означає, що в цьому файлі записуються всі перехоплені виклики методів обслуговування від мітки часу фреймворку 1368691248161 до -моменту зупинки системи.

Реалізація архітектури LogOs на OSGi Система LogOs працює в системі на основі OSGi framework (рис. 2.12). Система LogOs використовує пакет EventHook для створення проксі-сервера перехоплення. Після створення проксі служби для однієї зареєстрованої служби інші служби повинні викликати її методи

обслуговування через цей вбудований проксі служби. Цей виклик є непрямим. Цей проксі-сервер служби може фіксувати події журналу.

Система LogOs може спостерігати за всіма викликаними методами служби, які визначені в інтерфейс служби з java-анотаціями, навіть якщо послуга, що використовується, не зареєстрована і з'явилася нова служба.

```
GLOBAL{
   VARIABLES{ ... }
  EVENTS{ ... }
 PROPERTY P1 {
    STATES{...}
   TRANSITIONS{ ... }
  }
 FOREACH (Object u ){
      VARIABLES{ ... }
     EVENTS{
          %% Property designer
             needs to
          %% express how to
             retrieve the
          %% identifier:
          someEvent() = {*.method}
          ...
      }
      PROPERTY P2 {
         STATES{...}
         TRANSITIONS{ ... }
      }
 }
}
```

Рисунок 2.13 - Загальний файл з двома властивостями двох типів

Синтаксис Larva Для цього інструменту Larva опис властивостей контрольованої системи є ключовим моментом. Ця мова опису властивостей спрямована на опис кінцевого автомата (FSM) для перевірки запущеної системи програмного забезпечення. У файлі опису властивостей необхідно оголосити чотири частини: ПОДІЇ, ЗМІННІ, СТАНИ, ДІЇ ТРАНСЕКЦІЙ. Тут ми представляємо синтаксис кожної частини оголошення. Для полегшення розуміння принципу опису властивостей у Larva ми використовуємо приклад, щоб пояснити чотири частини декларації.

Загальну структуру файлу властивостей Larva наведено на рис. 2.13. Він показує файл, що містить дві властивості: глобальну та екземплярну.

Обидва інструменти (система LogOs і система Larva) корисні для відповідного середовища. Але для кожного середовища існують деякі недоліки. Система LogOs може фіксувати певні дії з динамічною системою. Але він не може перевірити, чи знята дія авторизована чи ні. Larva може перевіряти вказані властивості. Однак він не має жодних характеристик динамічної стійкості. У наступних двох розділах ми представимо нашу адаптацію системи LogOs та інструменту Larva для вирішення цих проблем.

2.8 Реєстрація специфікації надання послуг

Ми пропонуємо включити декларацію властивостей для моніторингу як частину пакетів OSGi , як показано на рис. 2.14. Дійсно, комплект OSGi — це архів, що містить чотири елементи: колекцію інтерфейсів, колекцію реалізацій служб, початковий код і POM.xml, який викликається під час завантаження або вивантаження комплекту. Завдяки архітектурі OSGi інтерфейси послуг, реалізації послуг, пакети та файл маніфесту POM можуть мати різні життєві цикли залежно від схеми розгортання, оскільки інтерфейси можуть розгортатися з пакетом, відмінним від того, що містить реалізацію служби.

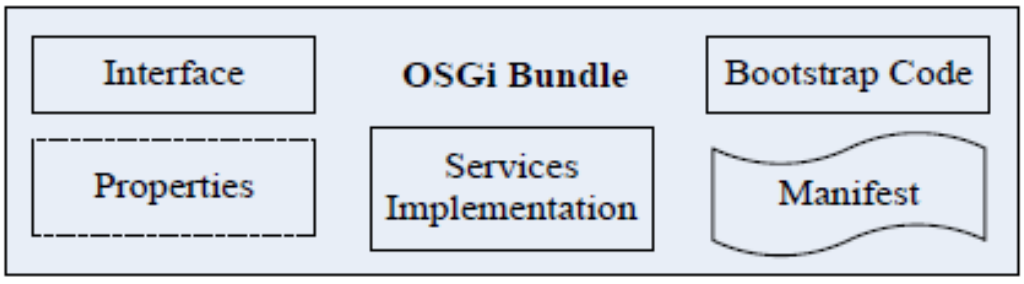


Рисунок 2.14 - Структура комплекту OSGi

Таким чином, ми пропонуємо зберегти ту саму філософію при наданні власності. Ми вважаємо, що вони можуть бути надані тим самим набором, що й реалізація, або іншим. Оскільки інтерфейси є специфікаціями типів служб, а OSGiLarva Class-Properties є поведінковими специфікаціями служб, має сенс

зіставити життєвий цикл властивостей класу з життєвим циклом властивостей інтерфейсів. З іншого боку, життєвий цикл властивостей екземпляра описує поведінку всієї взаємодії служби, і тому має сенс зіставити його життєвий цикл із життєвим циклом з'єднання клієнт-служба.

Сервісно-орієнтований підхід є парадигмою, що дозволяє внести динамічність у розвиток. Якщо у цього підходу багато переваг, є й нові проблеми, пов'язані зі зникненням служби. Окремий випадок заміни послуги часто вивчається, і існує багато пропозицій. Однак запропоновані рішення в основному є серверними і часто в контексті веб-сервісів. У цьому розділі ми пропонуємо клієнтський підхід до безпечного використання служби, щоб дозволити заміну служби без будь-якого перезапуску клієнта та без будь-яких припущень про зовнішні служби. Наша пропозиція базується на транзакційному підході, визначеному для автоматичної та динамічної заміни послуг шляхом збереження поточного виконання та запропонованих зібраних даних.

Коли архітектурні рішення були побудовані, то їх можна порівнювати за допомогою попраного співставлення схожих архітектур які тим чи іншим чином виконують такий сами функціонал.

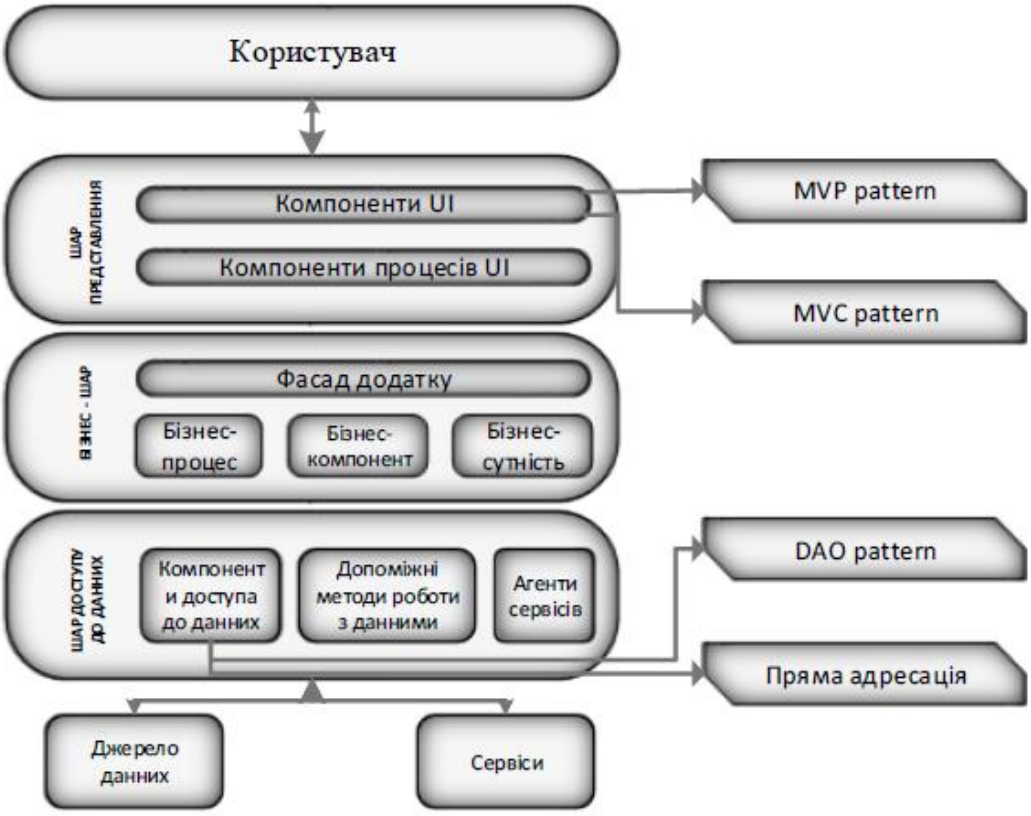
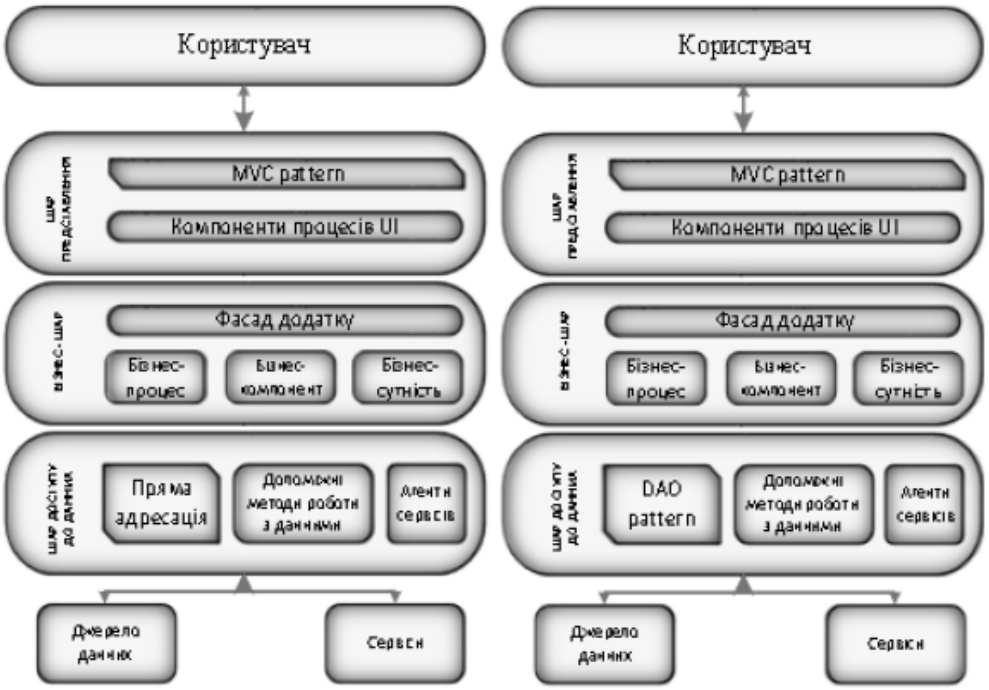


Рис. 2.15. Належність шаблонів проектування до певної архітектури



а) б)

Рис. 2.16. Альтернативні рішення використання паттернів:

а) MVC – здійснення прямої адресації; б) MVC – DAO

в) г)

а) б)

Рис. 2.17. Альтернативні рішення використання паттернів:
а) MVP – здійснення прямої адресації. б) MVP – DAO

Висновки до розділу

В даному розділі пропонуємо застосувати динамічний підхід до систем моніторингу виконання шляхом вставки моніторів у точці зв’язування клієнт-сервер, а не «статично» під час компіляції чи завантаження. Це означає, що як прив’язки до послуг, так і прив’язки для моніторингу поведінки є динамічними та слабко пов’язаними, таким чином підтримуючи заміну послуг.

РОЗДІЛ 3. Реалізація рівнів безпечної імплементації служб і паттернів в архітектурі програмного забезпечення

3.1 Опис пропонованого рішення побудови архітектури

Динамічний сервіс-орієнтований підхід є парадигмою, що вводить слабий зв'язок в архітектури програмного забезпечення. Розробник може просто вибрати потрібний опис API послуги та розробляти програмне забезпечення, не знаючи, яка реалізація - зрештою буде встановлена на кінцевій клієнтській системі. На даний момент найбільш популярним використанням цього підходу є веб-сервіси, Enterprise Java Beans (EJB), системи Android і фреймворк OSGi. Основні дослідження присвячені веб-сервісам, але з точки зору сервера [46]. Це означає, що постачальник послуг може робити багато припущень щодо наданих послуг з метою, щоб заміна послуги була здійснена без будь-яких наслідків для клієнта, навіть якщо послуга повна . Сервіси з повним станом підтримують внутрішній стан під час послідовних викликів від одного запитувача.

Ми пропонуємо вивчити точку зору клієнта, і ми базуємо нашу пропозицію на основі OSGi [9]. Ми зосереджуємося на випадку мобільної платформи з OSGi, яка може виявити або втрачати з'єднання з деякими постачальниками послуг. У такому випадку запитана клієнтом послуга може бути втрачена під час використання. Отже, ми не можемо зробити жодної жорсткої гіпотези щодо терміну служби послуг, але ми можемо запропонувати деякі хороші PR- практики в розвитку клієнта , щоб бути стійкими до заміни послуг. Заміна послуг добре відома як програмна техніка самовідновлення [37]. У цій роботі ми пропонуємо рішення, щоб зробити служби більш самовідновлюваними на стороні клієнта без зміни коду клієнта під час вивантаження служби.

Якщо сервіс розвантажений, то основні проблеми:

1. виявити службове розвантаження;
2. вибрати нову послугу;

3. завантажити новий сервіс, зберігаючи внутрішній стан незавантаженого.

В цьому розділі ми пропонуємо підхід безпечного використання послуг (SSU) для розробки клієнта, натхненний транзакційним підходом паралельних систем. Ми розглянемо задачу виявлення незавантаженого сервісу та завантаження одного з нових сервісів. У нашій пропозиції ми спочатку розглянемо простий випадок використання однієї послуги, перш ніж запровадити заміну послуги під час використання набору повних служб.

Якщо програмне забезпечення розроблено з використанням правильного підходу SSU, ми гарантуємо, що воно може відповідно до своїх уподобань:

1. отримувати активне сповіщення про вивантаження за допомогою спеціального винятку,

2. продовжити його виконання новою службою, яка була автоматично замінена, навіть якщо ця служба є повною, з дуже невеликими витратами коду для написання.

Ми також стверджуємо, що вартість розробки низька в порівнянні з вартістю розробки подібного програмного забезпечення з такими ж можливостями, але розробленого без використання підходу SSU. Нарешті, ми покажемо, що наш підхід не обмежує виразність розробленого програмного забезпечення, що означає, що кожна програма, яка використовує сервіс, може бути переписана для використання запропонованого підходу SSU.

У Розділі ми представили приклад моделі динамічної системи, яка контролюється відповідно до нашої пропозиції. Тут ми будемо використовувати систему бронювання авіакомпаній, щоб висловити ідеї, запропоновані в цьому розділі.

Правильне використання цієї системи бронювання авіакомпаній виглядає так: крок 1 вибирає політ, крок 2 оплачує бронювання і останній крок підтверджує це бронювання авіакомпанії. Цей приклад (рис. 3.1) ілюструє той факт, що: якщо один клієнт використовуватиме кілька служб одночасно, перед викликом confirm(...), посилання на службу S 1 стане застарілою, якщо Si буде

незареєстровано. Щоб уникнути цієї проблеми, ми пропонуємо створювати виняток або замінювати його новою службою, коли виникає застаріле посилання.

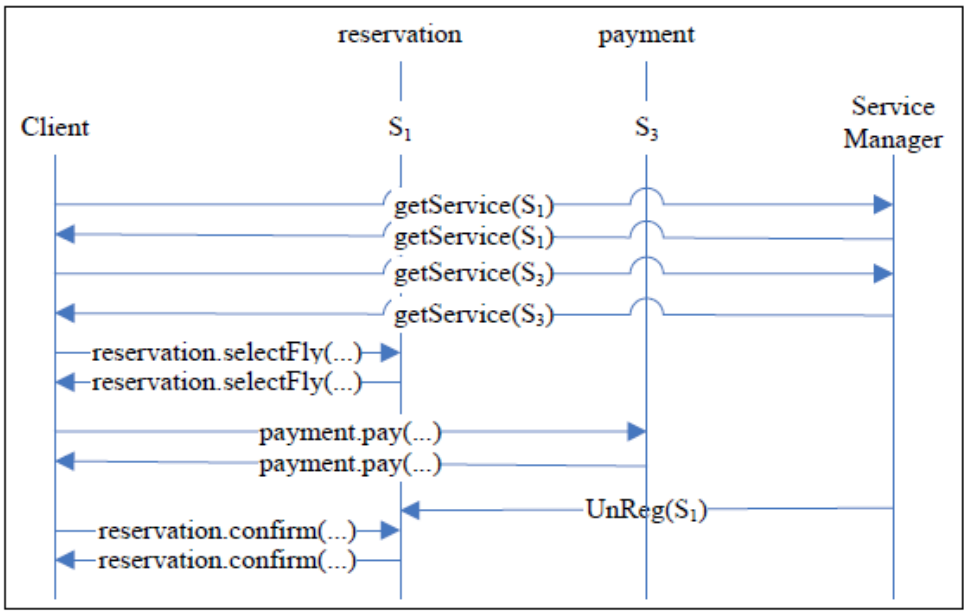


Рисунок 3.1 - Приклад виникнення застарілого посилання в архітектурі SOA

Ці пропозиції детально описано через два сценарії виконання цього прикладу на рис. 3.2 і рис. 3.3 .

У сценарії з рис. 3.2 S і скасував реєстрацію між двома викликами оплати (...) і підтвердження (...) , і не існує жодної служби, яка б це замінила. Під час виклику підтвердження запропонований рівень SSU відкочує resetPay ( ) , а потім створить виняток , щоб повідомити викликаному клієнту, що посилання на службу Si є застарілим.

Щоб проілюструвати другий підхід, заснований на заміні послуги, ми припустимо, що S1 та S2 реалізують бронювання, а S3 реалізує платіж у сценарії з рис. 3.3 . Потім ми висловлюємо наступне: на початку транзакції бронювання авіакомпанії клієнта нам спочатку потрібно створити проксі-сервери для двох інтерфейсів служби. Ці проксі-сервери використовуються для перевірки того, чи служба незареєстрована, і вжиття відповідних заходів для запобігання використанню застарілих посилань.

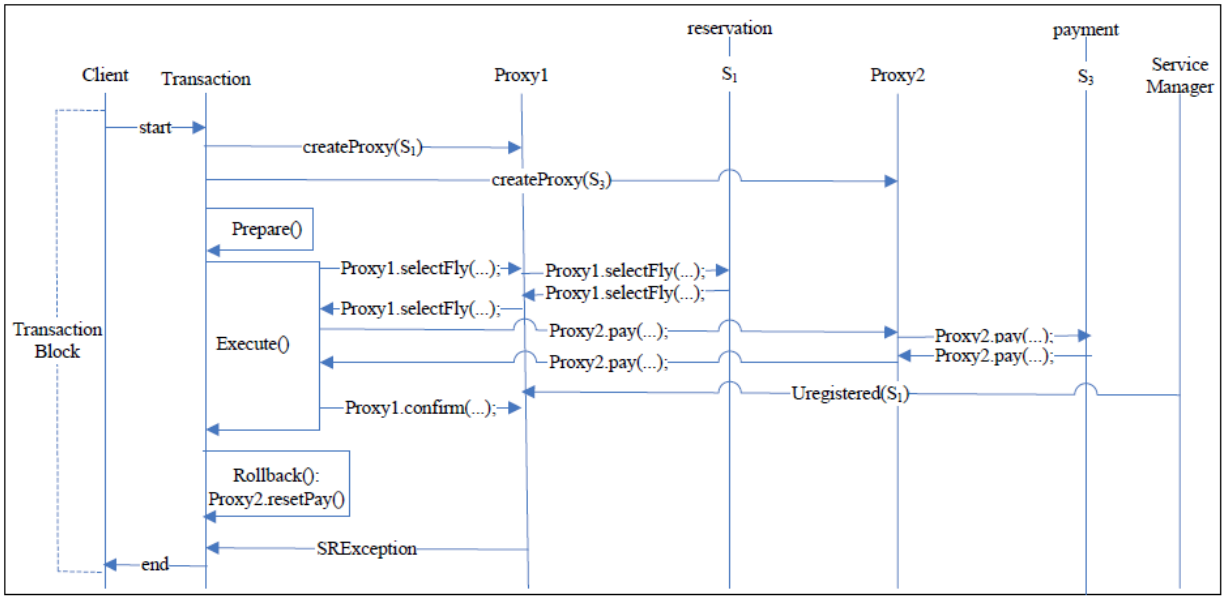


Рисунок 3.2 - Приклад сценарію з винятком для обробки застарілого посилання

Під час виконання транзакції, якщо S1 скасовано реєстрацію перед викликом confirm(...) , а потім запит на confirm(...) від клієнта надсилається на Proxyl , Proxyl може перевірити, чи поточна використана послуга (тобто S1) незареєстрований. Цей блок транзакцій запускає Rollback() , щоб повернути використані пов’язані служби (тобто виконати метод resetPay() S 3 через Proxy2 у цьому методі транзакції Rollback ()), а потім Proxyl виконує заміну (тобто S2 замінює незареєстрований S1). Після заміни послуги цей блок транзакцій повторно виконується від свого методу транзакцій Prepare() до кінця Finish() . Нарешті, цей блок транзакцій повертає клієнту результати від методу транзакції Execute() .

Цей приклад пояснює, що служба OSGi, яка працює з нашою пропозицією, може уникайте застарілих посилань. Транзакційний підхід може забезпечити узгодженість перехресного використання кількох послуг в одній обробці.

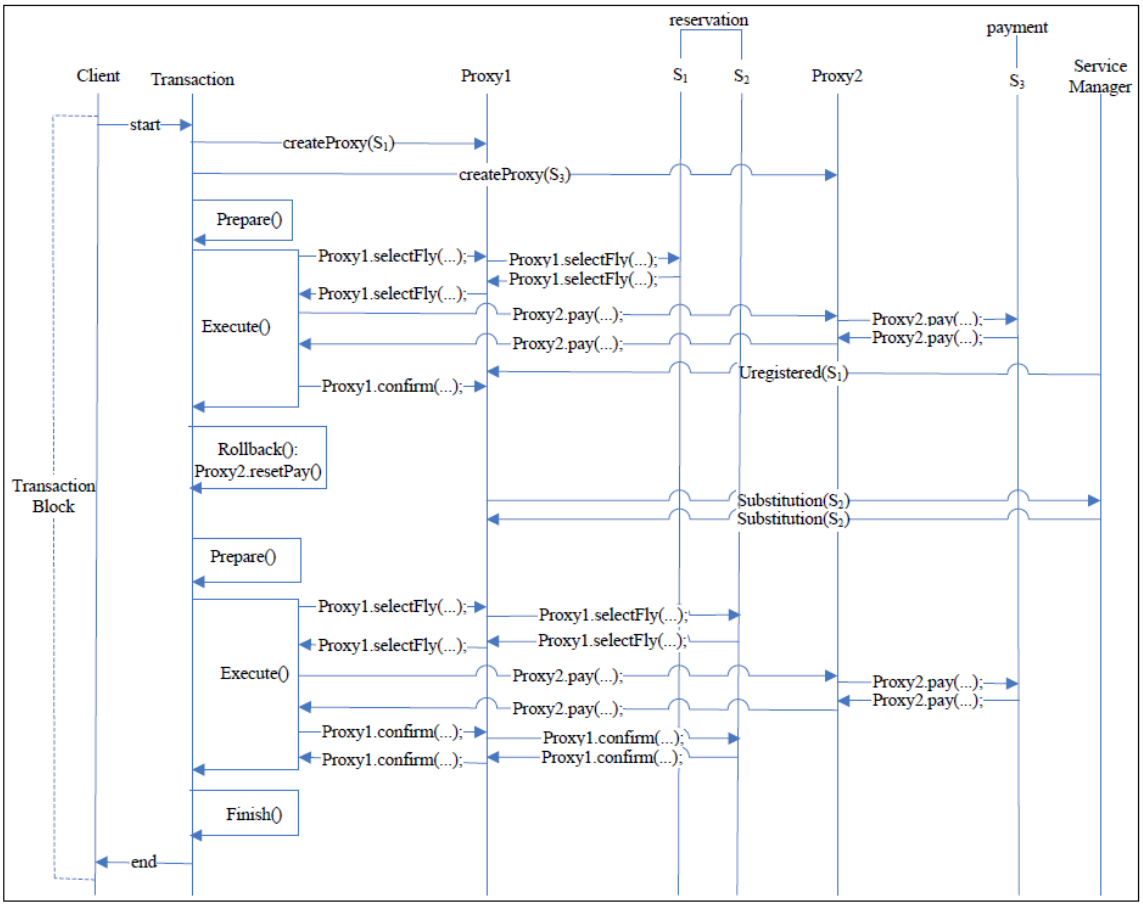


Рисунок 3.3 - Приклад сценарію із заміною послуги

Ми пропонуємо додати рівень SSU в структуру OSGi, щоб зробити системи більш стійкими до відмов. Цей рівень SSU використовувався клієнтами, щоб розпізнати службу розвантаження. Щоб описати, що має робити цей рівень SSU, ми спочатку представимо звичайні підходи до відмовостійких систем. Далі ми опишемо наші рішення для простого випадку, коли клієнт використовує один сервіс. Нарешті, ми розширюємо рішення, щоб врахувати випадок, коли одночасно використовуються кілька повних служб стану.

3.2 Опис відмовостійкості технології

Наша пропозиція спрямована на те, щоб система на основі OSGi автоматично обробляла динамічність служб, щоб уникнути посилань на нульовий вказівник і ефектів застарілих посилань під час виконання . У цьому розділі ми пояснимо, як використовувати відмовостійку технологію в системі на

основі OSGi для обробки виниклих помилок. Зазвичай відмовостійкі системи, виконання яких може продовжувати надавати коректне обслуговування, навіть якщо виникає збій. У такій системі перша проблема полягає в ідентифікації того, що сталася помилка [11]. У нашій пропозиції ми точно визначаємо, що є несправністю: розвантаження використаної послуги. І ми виявимо це за допомогою слухача, який додається під час реєстру служби.

Зазвичай існує три групи лікування для відновлення помилки [37]:

1. маскувати помилку;

2. відкат вперед у виконанні, доки не буде досягнуто стабільного стану;

3. щоб повернутися до попереднього стабільного стану та перезапустити виконання з нього.

Зазвичай механізм маски та помилки залежить від надлишкової інформації, - наданої системою. Оскільки ми не можемо мати його, ми зосередимося на двох інших видах лікування. Ми пропонуємо модель, пов'язану з останніми двома сімействами лікування. Щоб реалізувати механізм відкатування, коли служба зникає, ми пропонуємо створити виняток, який явно повідомляє клієнту, що послуга більше не доступна. Цей механізм схожий на сценарій, описаний на рис. 3.2 . Механізм спроби й уловлювання міг би тоді дозволити досягти нового стабільного стану вперед. Нарешті, щоб реалізувати механізм відкату, коли служба зникає, ми пропонуємо автоматичну заміну служби іншою, яка реалізує той самий інтерфейс служби. Ця заміна служби буде стійкою до повного стану, як описано в сценарії на рис. 3.3 . Дійсно, заміна служби не потребує перезапуску клієнта та будь-яких припущень про зовнішні служби через використання механізму транзакцій.

3.3 Інтерфейс виконання транзакційних блоків

Ми пропонуємо інтерфейс, який є службою OSGi для виконання транзакційних блоків. Цей інтерфейс виглядає наступним чином:

```
public interface TransactedServiceExecutor {
  public T executeInTransaction(
        TransactedExecution<T> execution,
        RetryPolicy retryPolicy)
      throws TransactedExecutionFailed;
}
```

Щоб реалізувати цей блок, нам потрібно мати транзакційне виконання для реалізації вказаної транзакції, а також потрібно мати політику повторних спроб для обробки дії відкоту від виконуваної транзакції. Отже, наступні два блоки коду щодо інтерфейсу TransactedExecution та інтерфейсу RetryPolicy можуть їх реалізувати.

Транзакційне виконання вказується через наступний інтерфейс

```
public interface TransactedExecution<T> {
  public void prepare();
  public T execute();
  public void finish();
  public void rollback();
}
```

Він використовує параметричний тип T , який є очікуваним типом поверненого значення успішного виконання транзакційного блоку.

Нарешті, інтерфейс RetryPolicy — це простий інтерфейс, який отримує сповіщення про потенційні застарілі помилки посилань і може, у свою чергу, вирішити, чи можна здійснити подальшу спробу . Його також можна використовувати для реалізації затримок між повторними розглядами. Прикладом би бути експоненціальною затримкою відстрочки протягом щонайбільше 10 виконань. Цей інтерфейс визначається наступним чином:

```
public interface RetryPolicy {
    public void notifyOf(Throwable throwable);
    public boolean shouldContinue();
 }
```

До речі, можливу політику "повторної спроби назавжди" можна реалізувати таким чином:

```
public class RetryForeverPolicy implements RetryPolicy {

    @Override
    public void notifyOf(Throwable throwable) { }

    @Override
    public boolean shouldContinue() {return true;}
}
```

Враховуючи деякі фіктивні сервісні інтерфейси SomeService та OtherService, реалізація транзакційного блоку може бути такою:

```
private class SomeTransaction
    implements TransactedExecution<Void> {

  @ServiceInjection public SomeService someService;

  @ServiceInjection(type = OtherService.class,
                    proxyType = MULTIPLE)
  public Set<OtherService> otherReferences;

  @Override public void prepare() { }
  @Override public <Void> Void execute() {
    for (OtherService s : otherReferences) {
      s.doThis(someService.doThat());
    }
}
  @Override public void finish() {
    someService.release();
```

У наведеному вище блоці коду демонструється виконання транзакційного блоку для кількох служб . Він складається з 4 кроків: подготвити(), виконати(), відкотити() і завершити(). Кожен крок має свою відповідальність у цьому блоці,

що описано на рис. 3.4. У execution() він виконує приблизно той самий метод, який викликають інші служби. Якщо ця транзакція завершена, цей викликаний метод обслуговування буде звільнено. Або транзакція скасує це виконання для всіх викликаних служб.

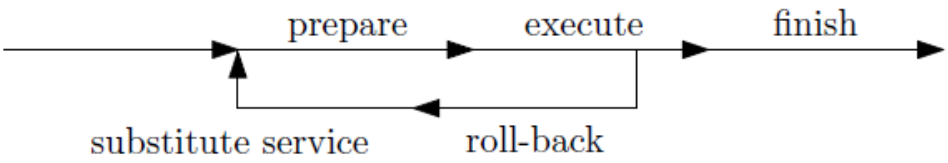


Рисунок 3.4 - Діаграма транзакцій для кількох послуг

Більш повний приклад приділяв би більше уваги підготовці (), roUback() і закінченню() . Анотації дозволені в Java з Java 1.5. Ми можемо використовувати анотацію Java , щоб визначити деякі поля або методи, які нам потрібні під час розробки. Поля, анотовані @ServiceInjection , надані цим API рівня SSU, додаються за допомогою проксі- серверів служби. Визначення цієї анотації таке:

```
@Retention(RUNTIME)
@Target(FIELD)
@Documented
public @interface ServiceInjection {
  Class<?> type() default ServiceInjection.class;
  String filter() default "";
  ProxyType proxyType() default SINGLE;
  ProxyMode proxyMode()
      default DISABLED_AFTER_UNREGISTERED;
  public static enum ProxyType {SINGLE, MULTIPLE}
}
```

Він використовується для налаштування способу налаштування проксі-серверів. Зокрема, у них можуть бути ввімкнені можливості перезавантаження служби, і вони можуть підтримувати одне посилання або набір екземплярів, як у випадку з набором otherReferences у попередньому прикладі. Також можна вказати фільтр служби OSGi.

Ми використовували оптимістичний підхід. Маючи сервісні проксі-сервери, введені в транзакційні блоки, ми могли б скористатися ними, щоб виконати гігантське блокування, що охоплює весь час виконання транзакції. Дійсно, можна заблокувати потік, який сповіщає про те, що служба збирається зникнути, таким чином зберігаючи посилання дійсним, доки всі отримувачі не будуть повідомлені. Такий підхід дозволить уникнути необхідності відкатів за більшої вартості обмеження паралелізму та порушення вимог інфраструктури OSGi, які обробники сповіщень про події служби не повинні блокувати [9].

У цьому розділі ми запропонували підхід і інструмент, який можна безкоштовно завантажити, щоб повідомити службу про проблему застарілих посилань. Якщо програмне забезпечення розроблено з правильним використанням цього підходу SSU, ми гарантуємо, що воно може, відповідно до своїх уподобань:

1) бути активно поміченим про вивантаження за допомогою винятку,

2) продовжити своє виконання новою службою, автоматично заміненою , навіть якщо послуга є повноцінною.

Основними властивостями цього внеску є:

1) це рішення на стороні клієнта,

2) воно не робить жодних припущень щодо використовуваних послуг,

3) його можна використовувати, навіть якщо використовувані служби є послугами з повним станом.

Цей внесок заснований на тому, що клієнт-дизайнер знає, як користуватися бажаними послугами. Таким чином, ми не намагаємося обчислити, яка поведінка дозволена службою. Розробник клієнта має лише нормально використовувати службу та запропонувати послідовність відкоту до стабільного стану, перш ніж зробити ще одну спробу з іншою службою, у разі заміни.

Однак, якщо відкат виконується на зовнішній службі, відкат також буде виконано на самому клієнті, щоб підтримувати узгоджений глобальний стан. Оскільки така модель розробки є ризикованою, ми вважаємо за краще дати

наступну вказівку в розробці клієнта: не вносьте жодних змін у стан клієнта з його транзакційної частини.

Як пояснюється та проілюстровано в основній специфікації OSGi, правильно використовувати службу без застарілих посилань може бути дещо складно. Потім ми стверджуємо, що вартість розробки служби з використанням нашого підходу SSU становить низька в порівнянні з вартістю розробки аналогічного програмного забезпечення з такими ж можливостями, але розробленого без нашого підходу SSU. Однак, зважаючи на простоту проксі та той факт, що програми OSGi зазвичай не мають дуже частих викликів служб, ми можемо вважати незначними витрати часу на такий проксі. Найгірше, коли відбувається багато замін. Але в цей час вартість часу є компромісом для збереження безперервності у використанні клієнта. Дійсно, як пояснюється та проілюстровано в основній специфікації OSGi, правильно використовувати службу без застарілих посилань може бути трохи складно.

Крім того, ми не вводимо жодних обмежень щодо виразності послуг. Насправді, у гіршому випадку, ми можемо включити всю програму в одну транзакцію. Отже, якщо служба замінюється, то вся програма перезапускається та повністю виконується з новою службою. Отже, цей підхід SSU не додає жодних обмежень у розробці програмного забезпечення.

У попередніх розділах ми вперше представили OSGiLarva, систему моніторингу для систем на основі OSGi. По-друге, ми запровадили рівень безпечного використання служб (SSU), API, який допомагає розробляти клієнтські програми, які справляються із застарілими посиланнями.

3.4 Представлення стійкої архітектури системи моніторнгу

У цьому розділі ми пропонуємо нові системи моніторингу (NewMS) (рис. 3.5), які поєднують систему OSGiLarva та рівень SSU. Це дає змогу не лише підтримувати динамічність систем на OSGi, а й мати справу із застарілими посиланнями з рівня SSU. Монітор знаходиться між клієнтом і сервером і

вставляється під час зв'язування клієнт-сервер. NewMS зосереджується на таких принципах: стійкість до динамічності, комплексність і відмовостійкість. Оскільки NewMS розглядає деякі нові події властивостей і описи властивостей, пов'язані з обробкою застарілих посилань, вона виграє в моніторингу точності, коли виникають застарілі посилання.

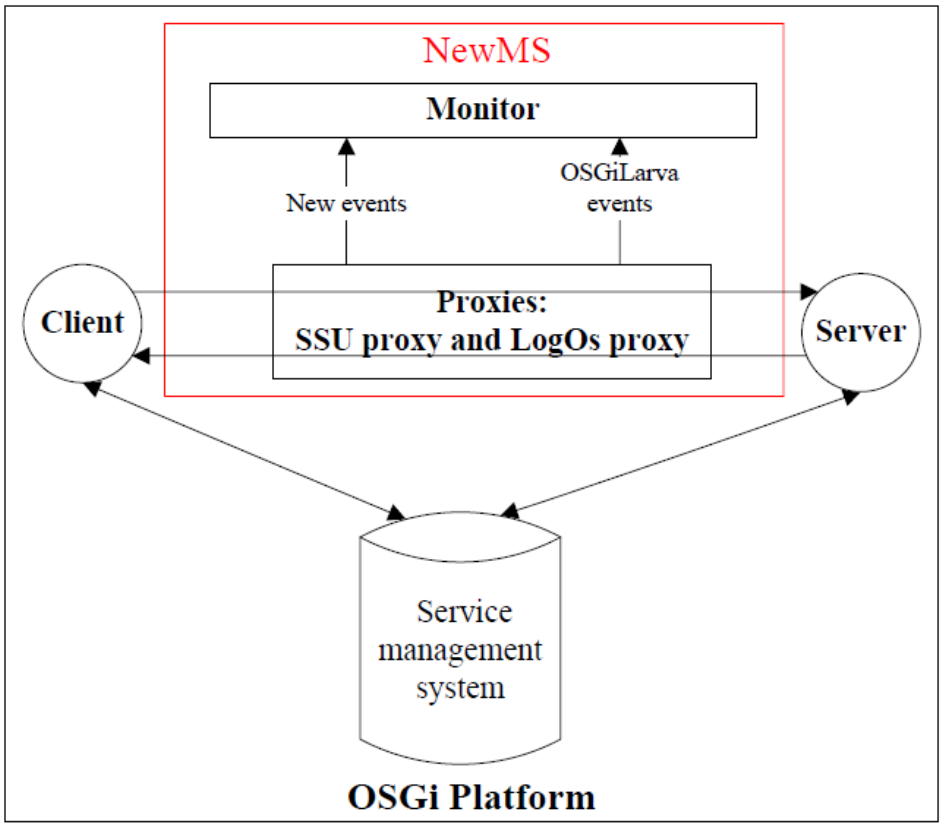


Рисунок 3.5 - Типова архітектура динамічної синтезованої системи моніторингу

Щоб досягти цього, існують деякі обмеження під час безпосереднього використання обох систем разом. Джерело цих обмежень в основному походить від використання двох проксі-серверів, з'єднаних перед однією службою: один проксі, який називається проксі-сервером LogOs, створюється для кожна зареєстрована служба. Інший проксі, який називається SSU proxy, створюється на стороні клієнта перед службою, яка справді є проксі-сервером LogOs. Клієнти спілкуються з проксі-сервером SSU, проксі-сервер SSU — із проксі-сервером

LogOs, а проксі-сервер LogOs — безпосередньо зі службою. Таким чином, SSU proxy не має прямого зв’язку зі службою.

Якщо ця служба незареєстрована, проксі-сервер SSU може здійснювати виклики методів за застарілим посиланням. І навпаки, якщо застаріле посилання виявлено проксі-сервером SSU, який створює виняток, проксі-сервер LogOs не може спостерігати цей виняток, оскільки виняток стався в системі рівня SSU.

Спочатку ми пояснюємо події властивостей, пов’язані з інтеграцією застарілих посилань у системі NewMS. Потім ми показуємо алгоритми для перекладу автоматів властивостей OSGiLarva в автомати властивостей NewMS.

У системі NewMS рівень SSU перевіряє під час кожного виклику методу служби, чи є посилання на службу застарілим чи ні. Якщо посилання застаріле, виклик призупиняється, а рівень SSU автоматично керує застарілим посиланням у трьох можливих випадках: заміна, відсутність альтернативної служби і відкотити не вдалося .

• Заміна: клієнт та інші служби, які наразі використовуються, відновлюються в тому самому статусі, що й до використання служби, що надає застаріле посилання. Рівень SSU виконав заміну послуги та заміну подія піднімається.

• Немає альтернативної служби : рівень SSU не знайшов альтернативної служби, але відкат клієнта пройшов добре, виняток SRException подія надсилається, і клієнту надсилається виняток застарілого посилання. У цьому випадку клієнт повинен знайти спосіб самостійно впоратися з відсутньою послугою.

• Помилка відкоту: клієнт зберігається в попередньому виклику відкоту, це означає, що він уже перебуває в застарілому керуванні посиланнями, і одночасно виникає новий. Ми не хочемо займатися цією справою. Потім ми повідомляємо клієнта, що у нас проблеми . Виняток RBException подія надсилається на монітор перед тим, як викликати виняток відкату.

Ці три події є підходом до розміщення автоматів моніторингу в найбільш точному положенні, коли виникає застаріле посилання.

Система NewMS забезпечує точне керування життєвим циклом відповідно до операцій OSGi. Ми розробляємо алгоритм (алгоритм 1, показаний на рис. 3.6), щоб автоматично транслювати автомат властивостей OSGiLarva в автомат властивостей NewMS.

Data: Let A = (S, s_0, B, V, v_0, Σ_M, Σ_F, δ) an OSGiLarva automaton which will be updated to generate a NewMS automaton. Some elements will not be changed: s_0, V, v_0, Σ_M.

step 1. Keep only transitions associated to methods call events in δ of A:

while *for each* t_1 *and DS, and* $(t_1=(s, e_1, l_1) \wedge t_1 \in \delta \wedge e_1 \in \Sigma_F \wedge DS= \{ds \mid \exists(c, a) ((c, a, ds) \in l_1)\})$, **do**

  $\delta=\delta-t_1$;

  while *for each* s_2, $s_2 \in DS$, **do**

    if s_2 *is not reachable from the initial state* s_0, **then**

      while *for each* t_2 *starting from* s_2, $(t_2=(s_2, e_2, l_2) \wedge t_2 \in \delta)$, **do**

        generate new transition like t_2 but starting from s and add it in δ, **by:**

        if *a transition* t_3 *with* e_2 *already exists* $(t_3=(s, e_2, l_3) \wedge t_3 \in \delta)$, **then**

          $\delta = (\delta - \{t_3\}) \cup \{(s, e_2, compose(l_3; l_2))\}$;

        else if *no transition like* t_3, **then**

          $\delta = \delta \cup \{(s, e_2, l_2)\}$;

        end

      end

    end

  end

end

step 2. delete all framework events names from Σ_F and add the three new property events names in Σ_F: $\Sigma_F = \{SRException, RBException, Substitution\}$;

step 3. add two new bad states in S and B of A:

$S = S \cup \{RBExp, SRExp\}$;

$B = B \cup \{RBExp, SRExp\}$;

step 4. From all non bad state, if "RBException" event occurs, it reaches state RBExp.

$\delta = \delta \cup (S - B) \times \{RBException\} \times [(true, nop, RBExp)]$;

step 5. From all non bad state, if "SRException" event occurs, it reaches state SRExp.

$\delta = \delta \cup (S - B) \times \{SRException\} \times [(true, nop, SRExp)]$;

step 6. From all non bad state, if "Substitution" event occurs, it reaches the initial state s_0.

$\delta = \delta \cup (S - B) \times \{Substitution\} \times [(true, nop, s_0)]$;

Рисунок 3.6 - Створення NewMS з автоматів OSGiLarva (Алгоритм 1)

Оскільки три додаткові події властивостей NewMS точніші, ніж події рамки OSGiLarva, ми видаляємо всі переходи, пов’язані з подіями рамки OSGi, а потім додаємо три функції переходу, пов’язані з новими визначеними подіями властивостей у автоматах властивостей NewMS. У цьому алгоритмі деякі згенеровані переходи позначаються як true і nop. Вони відповідають переходам, які завжди перетинаються без умови і без дії.

Функція compose(l1, l2) на алгоритмі 2 на рис. 3.7 використовується для об'єднання переходів, що починаються з того самого стану та запускаються тією самою подією.

Data: l_1 and l_2 are two lists of transition labels, where each label is a 3-tuple composed by a condition, an action and a destination state,

step 1. push bad states at the end of each list, by keeping original behaviors with the following method on each list:

while *there exists a location i in the list l such as state* s_i *is a bad state and* $s_{(i+1)}$ *is not a bad state,* **do**

swap labels and enforce condition of the pulled label.

For instance, this list

$l=\{(c_0, a_0, s_0), \cdots, (c_i, a_i, s_i), (c_{(i+1)}, a_{(i+1)}, s_{(i+1)}), \cdots\}$,

is transformed into

$l=\{(c_0, a_0, s_0), \cdots, ((c_{(i+1)} \wedge \neg c_i), a_{(i+1)}, s_{(i+1)}), (c_i, a_i, s_i), \cdots\}$,

end

Step 2. merge both lists by keeping the following global structure:

compose(l_1,l_2)= {(sub-list of l_2 without bad states);
(sub-list of l_1 without bad states);
(sub-list of l_2 with bad states);
(sub-list of l_1 with bad states)}

end

Рисунок 3.7 - Compose (l1, l2): створює два нові списки переходів (Алгоритм 2)

Дійсно, коли вводиться перехід від s1 до s2 , щоб перейти через видалений перехід, пов'язаний із подією рамки, новий перехід відповідає події, яка вже запущена. У такому випадку ми пропонуємо використовувати дозволену евристику злиття, засновану на політиці «звичайна поведінка спочатку». Це означає, що ми надаємо пріоритет поведінці, яка досягає непоганих станів. У цій політиці ми намагатимемося дотримуватися наступних вимог :

1. зберігати внутрішню поведінку (порядок) кожного списку;

2. нормальні стани l1 з його нормальною поведінкою мають пріоритет над l2 ;

3. усі погані стани l1 висуваються після нормальних станів l1;

Перший крок цього алгоритму натхненний алгоритмом бульбашкового сортування , щоб мати звичайні стани перед поганими в кожному списку, але

зберігаючи їхній пріоритет. На другому кроці обидва списки складаються разом за частиною непоганих станів і частиною поганих станів.

Ці перекладені автомати властивостей можуть бути реалізовані мовою опису властивостей OSGiLarva для моніторингу систем на основі OSGi. У наступному розділі ми наведемо приклад, щоб показати переклад за допомогою алгоритмів.

Щоб показати переклад з автомата OSGiLarva на автомат NewMS, ми представляємо приклад перекладу в цьому розділі. Система бронювання авіакомпаній зі специфікацією OSGiLarva є прикладом, який можна використовувати тут. Алгоритм 1 застосовано для трансляції автомата A OSGiLarva в автомат властивостей NewMS A' (рис. 3.8). Рис. 3.9 відповідає графічному зображенню цього перекладу.

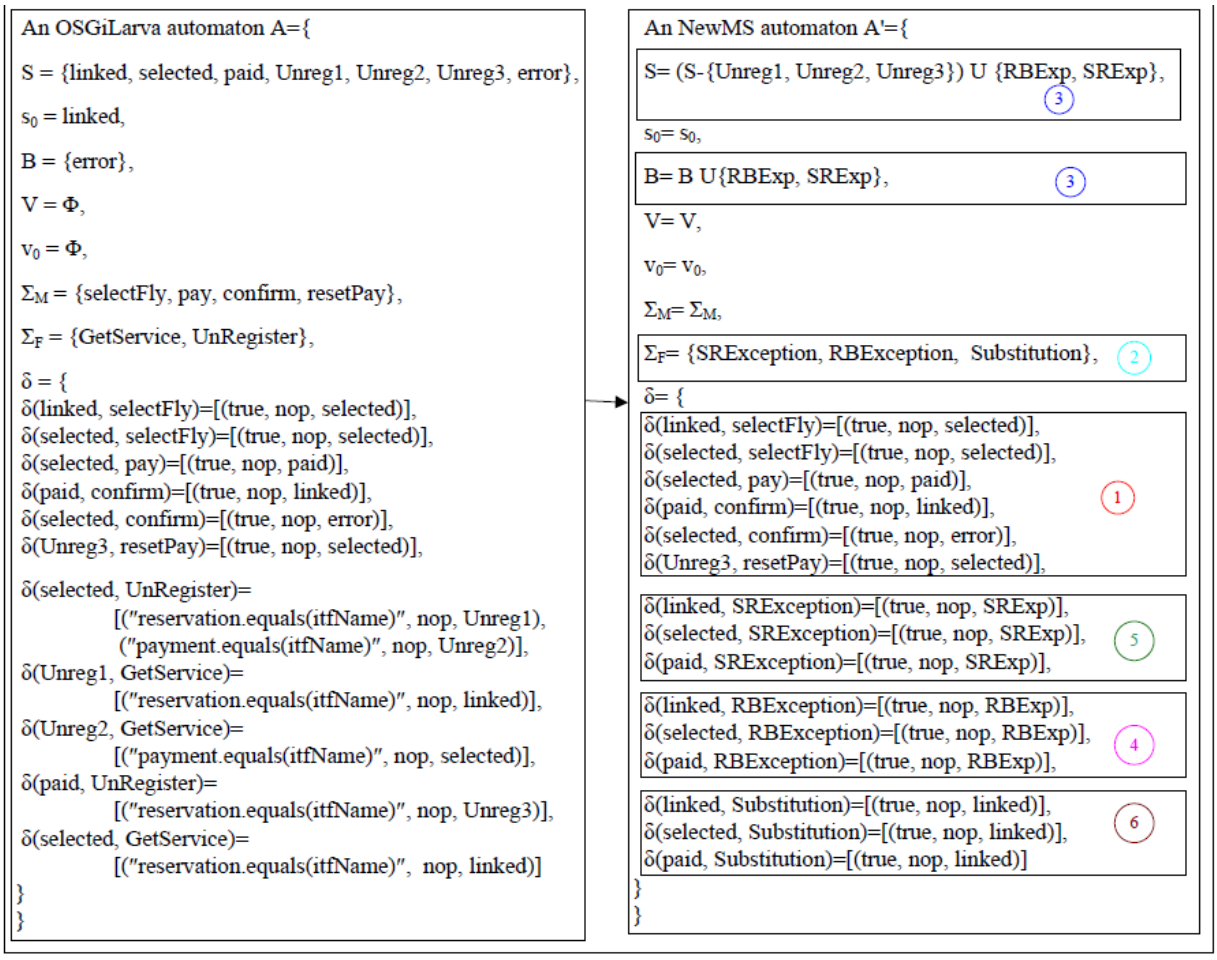
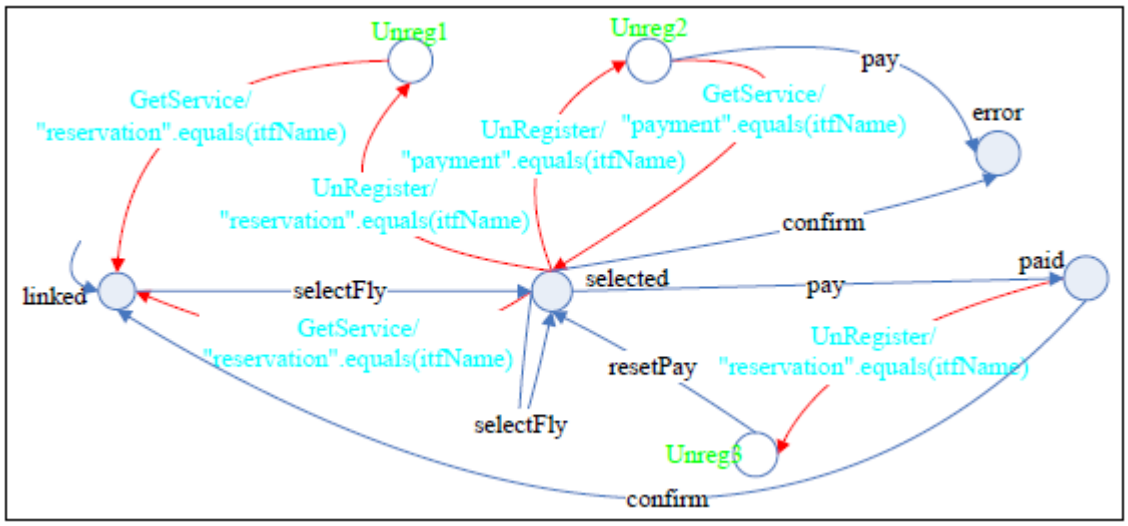
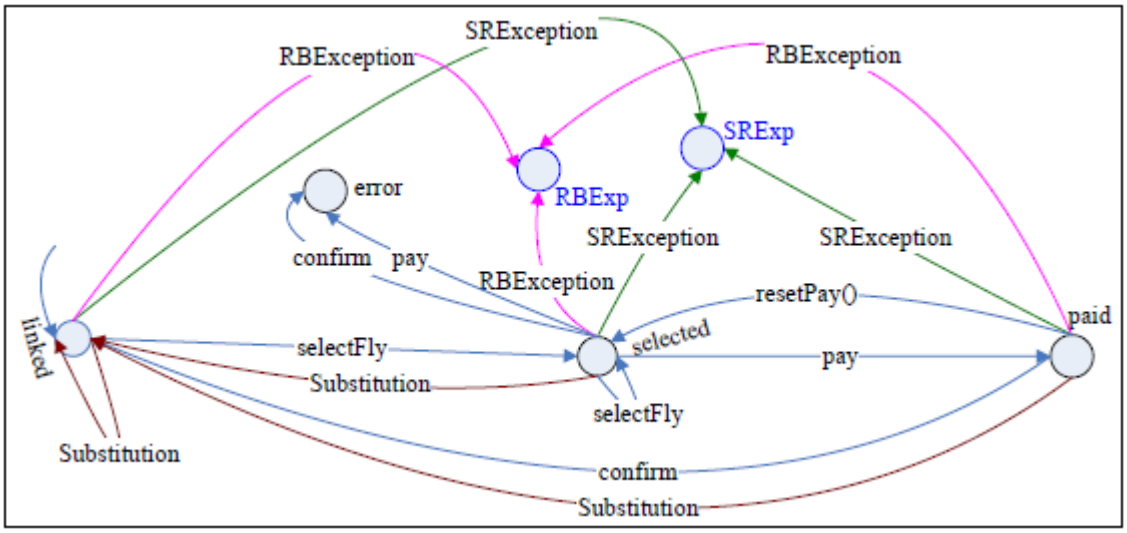


Рисунок 3.8 - Приклад кроків алгоритму з OSGiLarva на NewMS автомат

Під час перевірки виконання нашим монітором, якщо кінцевий стан переходу є поганим (тобто стани «помилка», «RBExp» і «SRExp» на рис. 3.8 і рис. 3.9), ця траса є закінчення помилки. Ідентифікація неправильної поведінки записується в журнал NewMS. Якщо всі стани призначення є непоганими, це трасування є правильним щодо властивості.



а) Автомат OSGiLarva властивості бронювання авіакомпаній



б) Перекладений автомат NewMS властивості бронювання авіакомпаній

Рисунок 3.9 - Трансляція автомата A OSGiLarva в автомат A' NewMS

На рис. 3.8 і 3.9 ми могли спостерігати, що немає ніякого вибуху розміру властивості, хоча ми додали деякі нові події та нові стани. Дійсно, кількість доданих переходів зростає лінійно з кількістю непоганих стани у властивості, а розмір створеного автомата зменшується через видалення рамкових переходів.

OSGiLarva може перевірити незареєстровану, зареєстровану та завантажену службу, але не впевнений, чи є поточне використане посилання на службу застарілою чи ні, і чи є служби, що використовуються, узгодженими чи ні. І навпаки, NewMS гарантує їх усі, оскільки рівень SSU автоматично керує послугами для клієнтів. Крім того, нові випадки (тобто Заміна, Безальтернативне обслуговування та Відкат не вдалося) , отримані рівнем SSU, роблять властивості більш точними та дозволяють виявляти більше деталей у NewMS, ніж у OSGiLarva.

Система OSGiLarva-SSU++ (рис. 3.10) інтегрує рівень SSU і системи OSGiLarva та реалізує архітектуру NewMS. Ця реалізація в основному зосереджена на системі перевірки власності та сумісній роботі системи LogOs і робочих проксі-серверів рівня SSU.

Ми запропонували алгоритм для створення автоматів властивостей NewMS з автоматів властивостей OSGiLarva. Ці згенеровані автомати властивостей NewMS можна описати у файлі властивостей мовою опису властивостей OSGiLarva.

Починаючи відстежувану систему, цей файл властивостей NewMS має бути скомпільований нашим адаптованим компілятором Larva для створення системи перевірки властивостей (PVS).

Коли клієнт використовує контрольовані служби, PVS починає перевіряти визначені події, які відповідають одному або декільком викликам методу від цього клієнта.

Виклики методів клієнта згруповані в блоці транзакцій. Конструкція цього блоку автоматично керує зв’язком між клієнтом і сервером через проксі SSU і проксі LogOs. Оскільки проксі-сервер LogOs створюється під час реєстрації служби, а проксі-сервер SSU створюється перед використанням зареєстрованої

служби, проксі-сервер SSU створюється після проксі-сервера LogOs. Проксі SSU активується перед проксі LogOs під час використання послуг клієнтами. Наприклад, коли клієнт викликає метод відстежуваної служби, його проксі SSU спочатку перехоплює цей виклик.

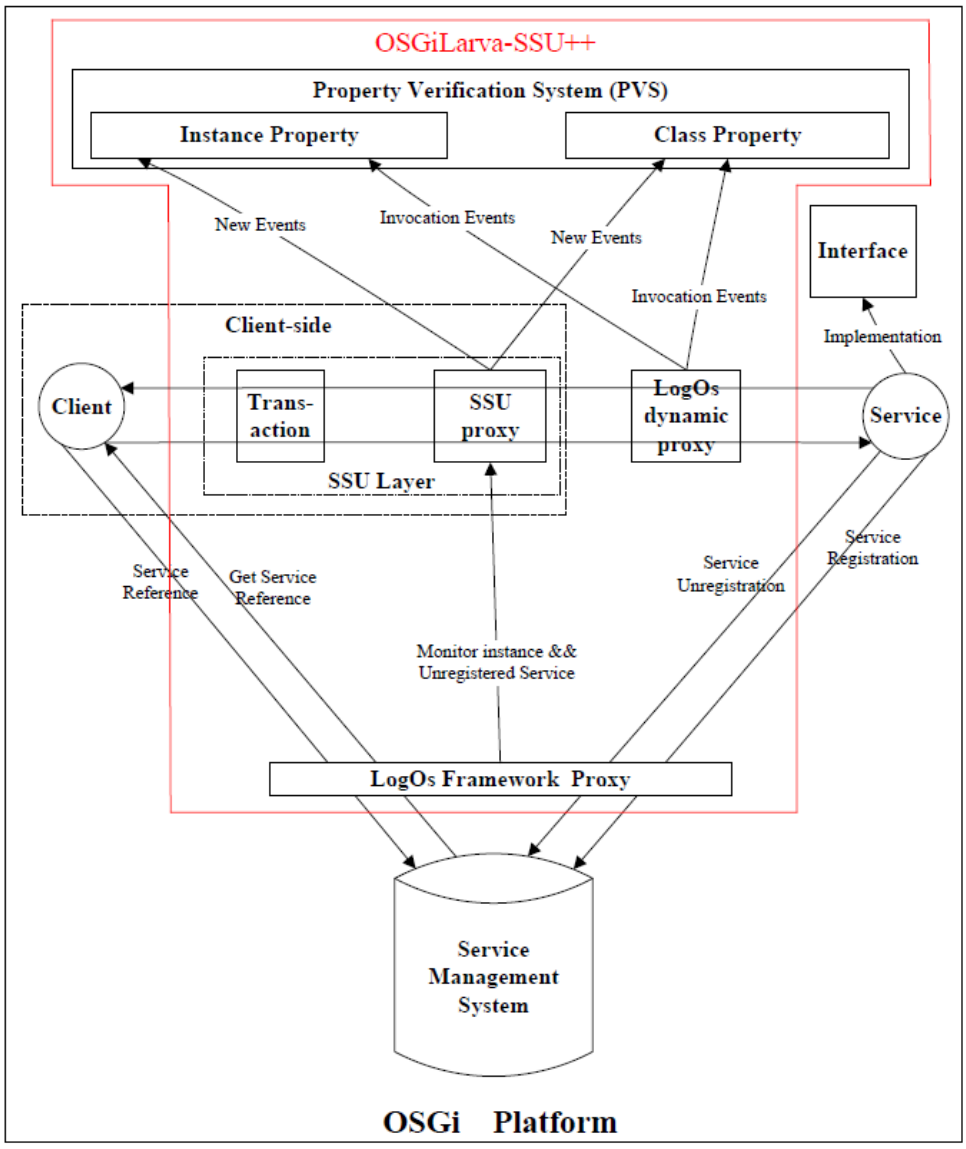


Рисунок 3.10 - Реалізація динамічної синтезованої системи моніторингу

Оскільки проксі-сервер рівня SSU ідентифікує справжню службу за допомогою проксі-сервера LogOs, на який посилається, і щоб уникнути використання клієнтами застарілих посилань у проксі-сервері рівня SSU, щоразу, коли служба скасовується з реєстрації, проксі-сервер LogOs сповіщає існуючі

рівні SSU. Отже, коли відбувається виклик методу служби, проксі-сервер рівня SSU спочатку перевіряє, чи незареєстрована послуга, що використовує:

1. Незареєстрований: Якщо послуга, що використовується, не зареєстрована, згідно з політикою виконується спроба заміни та відповідна нова подія (тобто Заміна або SRException або RBException) надсилається з його проксі SSU до PVS.

2. Зареєстровано: Якщо послугу, що використовує, зареєстровано, проксі-сервер SSU обробляє цей запит через проксі-сервер LogOs, проксі-сервер LogOs безпосередньо запитує його на сервер, і цей запит надсилається з проксі-сервера LogOs до PVS.

Нарешті, записи моніторингу надсилаються користувачам у реальному часі з PVS.У цьому розділі ми описали систему NewMS. Він поєднує в собі систему OSGiLarva і рівень SSU. Основні риси – динамічність, стійкість , комплексність і відмовостійкість успадковані від цих двох систем. Інтеграція може спостерігати три можливі випадки використання незареєстрованої послуги, наприклад: заміна, відсутність альтернативної служби і Помилка відкоту. NewMS може керувати трьома випадками та перевіряти відповідну подію за допомогою описаних автоматів властивостей.Ми також розробляємо алгоритм для автоматичного перекладу властивості OSGiLarva au tomata в автомати властивостей NewMS.

Нарешті, у порівнянні з системою моніторингу OSGiLarva, основними перевагами системи NewMS є точність інтерпретації, щоб перевірити, чи є посилання застарілим чи ні.

Висновки до розділу

В даному розділі описуються основні пропозиції щодо системи NewMS. Потім ми надаємо вказівки щодо реалізації архітектури NewMS з розв’язанням цих обмежень.

ВИСНОВКИ

В поданій роботі виконано дослідження та порівняльний аналіз архітектурних рішень на основі паттернів. В ході дослідження отримані результати виявились важливими для сучасної програмної індустрії, де зростаюча складність та швидкі технологічні зміни вимагають ефективних архітектурних рішень. ході дослідження були досягнуті наступні висновки:

- вибрані архітектурні рішення на основі паттернів проектування виявилися ефективними та придатними для розв'язання ряду завдань.

- застосування паттернів проектування сприяє покращенню якості, гнучкості та стабільності програмних продуктів.

- реалізовані рішення дозволяють оптимізувати витрати на розробку та підтримку, що важливо для підприємств та команд розробників.

Дослідження призводить до важливих висновків для архітекторів, розробників та керівників проектів, які прагнуть покращити якість та результативність своїх програмних продуктів.

В цілому, робота вносить важливий внесок у розуміння та застосування архітектурних рішень на основі паттернів проектування в програмному інженерії.

З урахуванням швидко змінюваного середовища програмного забезпечення, важливо відслідковувати актуальність та доповнювати отримані результати у майбутньому.

Основною метою дослідження було виявити, як різні паттерни проектування впливають на розробку, масштабованість, підтримку та ефективність програмного забезпечення. Результатом практичного значення отриманих результатів полягає в їхньому застосуванні для покращення архітектур програмних продуктів, раціонального використання ресурсів та підвищення загальної якості та ефективності розробки програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Open Service Gateway Initiative (OSGi), http://www.osgi.org/.

2. Javascript object notation web service protocol. http://en.wikipedia.org/wiki/JSON-WSP.

3. Web service description language: Wsdl 1.1, 2001.

4. Simple object access protocol:SOAP 1.2. W3C, 2003.

5. Transactional systems. In Reliable Distributed Systems, pages 509–528. Springer New York, 2005.

6. An autonomic approach to offer services in OSGi-based home gateways. Computer Communications, 31(13):3049 – 3058, 2008. Special Issue:Self-organization and self-management in communications as applied to autonomic networks.

7. H. Ahn, H. Oh, and J. Hong. Towards Reliable OSGi Operating Framework and Applications. Journal of Information Science and Engineering, 23(5):1379, 2007.

8. OSGi Alliance. About the OSGi service platform. Technical report, June 7 2007.

9. The OSGi Alliance. OSGi service platform core specification. Revision 4.1, April 2007.

10. Sven Apel and D. Batory. How AspectJ is used: an analysis of eleven AspectJ programs. Journal of Object Technology, 9:117–142, 2010.

11. A. Aviziens. Fault-tolerant systems. IEEE Trans. Comput., 25(12):1304–1312, December 1976.

12. Stefan Axelsson, Ulf Lindqvist, and Ulf Gustafson. An approach to UNIX security logging. In 21st National Information Systems Security Conference, pages 62–75, 1998.

13. Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In Proceedings of the IEEE International Conference on Web Services, ICWS ’06, pages 63–71. IEEE Computer Society, 2006.

14. Luciano Baresi, Domenico Bianculli, Carlo Ghezzi, Sam Guinea, and Paola Spoletini. Validation of web service compositions. IET Software, 1(6):219–232, 2007.

15. Mike Barnett, Robert DeLine, Manuel Fahndrich, Bart Jacobs, K.RustanM. Leino, Wolfram Schulte, and Herman Venter. The spec# programming system: Challenges and directions. In Bertrand Meyer and Jim Woodcock, editors, Verified Software: Theories, Tools, Experiments, volume 4171 of Lecture Notes in Computer Science, pages 144–152. Springer Berlin Heidelberg, 2008.

16. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: an overview. In Proceedings of the 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS' 04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

17. C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Reconfiguration Service for CORBA. International Conference of Configurable Distributed Systems, 1998.

18. JanOlaf Blech, Yliès Falcone, Harald Rueß, and Bernhard Schätz. Behavioral specification based runtime monitors for OSGi services. In Tiziana Margaria and Bernhard Steffen, editors, Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, volume 7609 of Lecture Notes in Computer Science, pages 405–419. Springer Berlin Heidelberg, 2012.

19. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf., 7(3):212–232, June 2005.

20. Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. Journal of Systems and Software, 81(4):463 – 480, April 2008.

21. Humberto Cervantes and Richard S. Hall. Automating Service Dependency Management in a Service-Oriented Component Model. In CBSE, 2003.

22. Patrice Chalin, JosephR. Kiniry, GaryT. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and esc/java2. In FrankS. Boer, MarcelloM. Bonsangue, Susanne Graf, and Willem-Paul Roever, editors, Formal

Methods for Components and Objects, volume 4111 of Lecture Notes in Computer Science, pages 342–363. Springer Berlin Heidelberg, 2006.

23. Mala chandra. Seamless mobility and OSGi service platform. Technical report, Client Application and Architecture Motorola, Inc., 2004.

24. Feng Chen. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In Electronic Notes in Theoretical Computer Science, pages 106–125. Elsevier Science, 2003.

25. Feng Chen and Grigore Rosu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In Nicolas Halbwachs and Lenore D. Zuck, editors, Tools and Algorithms for the Construction and Analysis of Systems, volume 3440 of Lecture Notes in Computer Science, pages 546–550. Springer Berlin Heidelberg, 2005.

26. Hao-Chung Cheng, Wei-Tsong Lee, Xin-Wen Wei, and Tian-Wen Sun. A novel service oriented architecture combined with cloud computing based on R-OSGi. In James J. (Jong Hyuk) Park, Qun Jin, Martin Sang-soo Yeo, and Bin Hu, editors, Human Centric Technology and Service in Smart Space, volume 182 of Lecture Notes in Electrical Engineering, pages 291–296. Springer Netherlands, 2012.

27. Yoonsik Cheon, Yoonsik Cheon, Gary T. Leavens, and Gary T. Leavens. A runtime assertion checker for the java modeling language (JML). In Proceedings of the International Conference on Software Engineering Research and Practice (SERP), Las Vegas, pages 322–328. CSREA Press, 2002.

28. Yoonsik Cheon, Gary T. Leavens, Yoonsik Cheon, and Gary T. Leavens. Asimple and practical approach to unit testing: The JML and JUnit way. In ECOOP 2002, volume 2374 of LNCS, pages 231–255. Springer Berlin Heidelberg, 2002.

29. Hsin Chou. Service oriented architecture for an overall radioactive waste package record management system. Progress in Nuclear Energy, 53:420–427, May 2011.

30. Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In Proceedings of the

20th international conference on Computer Aided Verification, CAV'08, pages 121 – 134, Berlin, Heidelberg, 2008. Springer-Verlag.

31. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic eventbased runtime monitoring of real-time and contextual properties. In FMICS, pages 135–149, 2008.

32. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva - safer monitoring of real-time java programs. In SEFM, 2009.

33. Christian Colombo, GordonJ. Pace, and Gerardo Schneider. Dynamic eventbased runtime monitoring of real-time and contextual properties. In Darren Cofer and Alessandro Fantechi, editors, Formal Methods for Industrial Critical Systems, volume 5596 of Lecture Notes in Computer Science, pages 135–149. Springer Berlin Heidelberg, 2009.

34. Yufang Dan, Nicolas Stouls, Christian Colombo, and Stéphane Frénot. OSGi-Larva: a monitoring framework supporting OSGi's dynamicity.

35. Yufang Dan, Nicolas Stouls, Stéphane Frénot, and Christian Colombo. A Monitoring Approach for Dynamic Service-Oriented Architecture Systems. In SERVICE COMPUTATION 2012: The Fourth International Conferences on Advanced Service Computing, pages 20–23, 2012.

36. B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Zohar Manna. Lola: runtime monitoring of synchronous systems. In Temporal Representation and Reasoning, 2005. TIME 2005. 12th International Symposium on, pages 166–174, 2005.

37. Rogerio de Lemos, Paulo Asterio de Castro Guerra, and Cecilia Mary Fischer Rubira. A Fault-Tolerant Architectural Approach for Dependable Systems. IEEE Software, 23:80–87, 2006.

38. Andre de Matos Pedro. Dynamic contracts for verification and enforcement of realtime systems properties. PhD thesis, Cister Research Unit - ISEP/IPP, September 17 2012.

39. Dionysis Athanasopoulos, Apostolos Zarras, and Valérie Issarny. Service Substitution Revisited. In 24th IEEE/ACM International Conference on Automated Software Engineering - ASE 2009, Auckland Nouvelle-Zélande, 11 2009. IEEE/ACM.

40. Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, PÅl Krogdahl, Dr Min Luo, and Tony Newling. Patterns: Service-Oriented Architecture and Web Services. Number SG24-6303-00. IBM Redbooks, ibm edition, April 29 2004.

41. Paul England. Practical Techniques for Operating System Attestation. In 1st international conference on Trusted Computing and Trust in Information Technologies (Trust'08), pages 1–13. Springer-Verlag, 2008.

42. Clement Escoffier, Richard S. Hall, and Philippe Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. In Services Computing, IEEE International Conference on, pages 474–481. IEEE Computer Society, 2007.

43. Roy Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine, 2000.

метадані

Заголовок

Порівняльний аналіз процесу імплементації архітектурних рішень при використанням різних паттернів проєктування

Автор

Чейпеш А.С. Науковий керівник / Експерт

підрозділ

King Danylo University

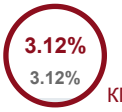
Тривога

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

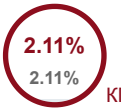
Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	13

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

7505
Кількість слів

57090
Кількість символів

Подібності за списком джерел

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

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	http://repository.ukd.edu.ua/bitstream/handle/123456789/388/%D0%94%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0%20%D1%80%D0%BE%D0%B1%D0%BE%D1%82%D0%B0%20%D0%9B%D0%B8%D1%82%D0%B2%D0%B0%D0%BA.pdf?sequence=1	91	1.21 %
2	http://repository.ukd.edu.ua/bitstream/handle/123456789/390/%D0%9C%D0%B0%D0%BD%D1%82%D1%83%D0%BB%D1%8F%D0%BA%20%D0%94.%D0%92.%20%D0%9A%D0%A0.pdf?sequence=1	37	0.49 %
3	http://repository.ukd.edu.ua/bitstream/handle/123456789/390/%D0%9C%D0%B0%D0%BD%D1%82%D1%83%D0%BB%D1%8F%D0%BA%20%D0%94.%D0%92.%20%D0%9A%D0%A0.pdf?sequence=1	30	0.40 %