УДК 004.4

М.М. Глибовець, С.С. Гороховський, І.М. Луценко

АНАЛІЗ КАРКАСІВ ЗБЕРІГАННЯ Й ВІДОБРАЖЕННЯ НА БАЗІ АСПЕКТНО-ОРІЄНТОВАНИХ ТЕХНОЛОГІЙ

Запропоновано набір рекомендацій використання ін'єкції залежностей і аспектно-орієнтованого програмування у розробці програмного забезпечення. Проаналізовано вплив їх використання на процес проектування, розробки, тестування, подальшої підтримки та архітектуру програмного продукту.

Вступ

Ін'єкція залежностей (ІЗ) і аспектно-орієнтоване програмування (АОП) дві технології, які набули значної популярності останнім часом. Хоча ці технології не пов'язані одна з одною, обидві мають спільне застосування щодо поліпшення проектування, впровадження та супроводу програмного забезпечення шляхом створення деякого базису, який зменшує зв'язок між об'єктами і, відповідно, дозволяє здійснити розбиття на окремі модулі так званої наскрізної функціональності [1 – 3].

Уточнимо, що ми розуміємо під висловом використанням ІЗ і АОП для проектування програмного забезпечення: в яких областях може ефективно застосовуватися; який вплив на дизайн програмної системи здійснюється з точки зору загальної модульності та архітектури програми.

Що означає вислів використання ІЗ і АОП для розробки програмного забезпечення? Нас буде цікавити визначення впливу зазначених підходів на побудову проекту з точки зору можливих змін і подальшої підтримки та організації тестування модулів, особливо автоматизованого.

Одним з найбільш використовуваних програмних каркасів з відкритим кодом для Java платформ є Spring Framework (SF). Цей каркас робить комерційну розробку програмних застосувань простішою, особливо у порівнянні з J2EE [4]. Програмне забезпечення (ПЗ), розроблене з його допомогою не повинно залежати від API програмного каркасу. Тому SF підтримує інтеграцію з цілим рядом інших рішень, таких як Hibernate та каркасами об'єктно-реляційного від ображення, а також веб-каркасами такими, як Struts. Це дозволяє використовувати тільки потрібні в даний момент розробки SF разом з іншими, вже існуючими, рішеннями.

SF надає чималу функціональність, розподілену на модулі, які можуть бути розглянуті як окремі повноцінні програмні каркаси. Серед таких модулів присутні: каркаси доступу до даних, керування транзакціями і безпеки підтримки віддаленого керування, обміну повідомлень і тестування, контейнер інверсії контролю. Останній і забезпечує конфігурацію компонентів програми за допомогою ІЗ і АОП.

Мета даної роботи – аналіз і оцінка наслідків введення і використання ІЗ та АОП з практичної точки зору в реальному проекті. Таким проектом будемо вважати обмежений як у часі, так ресурсами проект, що має нечітко сформульовані специфікації та мінливі вимоги.

1. Ін'єкції залежностей (Dependency Injection)

Ін'єкції залежностей походять з концепції інверсії контролю. Основна ідея інверсії контролю – це досягнення слабкішого зв'язування модулів і створення самих модулів системи придатними до повторного багаторазового використання шляхом делегування контейнерові або каркасу відповідальності за потік керування у застосуванні [5], тобто, інвертування потоку управління у порівнянні з традиційною архітектурою. Звідси і назва. Ін'єкція залежності (Dependency Injection)

© М.М. Глибовець, С.С. Гороховський, І.М. Луценко, 2013

є особливою формою інверсії контролю. Термін Dependency Injection був введений Мартіном Фаулером в 2004 році [6].

Використання ІЗ програмного каркасу призводить до усунення обов'язку створення, поєднання і компонування об'єктів клієнтами або самими службами. Вони будуть передані програмному каркасу. Тому потік керування інвертується. Замість відповідальності об'єкта за отримання необхідної йому служби з використанням паттерну *фабрика* [7], локатора служб (service locator) чи самостійного створення служби, об'єкт може мати аргумент, що містить посилання і автоматично встановлюється програмним каркасом під час створення об'єкта.

Існує три типи стилів ін'єкції залежностей: ін'єкція під час створення (constructor injection), відкладена ін'єкція (setter injection) та ін'єкція через інтерфейс (interface injection). Різниця між ними визначається в тому, як вони надають об'єкту посилання на його залежності. В першому випадку посилання на залежності надаються через конструктор класу, тобто під час створення об'єкта. У відкладеній ін'єкції об'єкт надає спеціальний метод для зберігання посилань на залежності, який використовує каркас. У останньому варіанті об'єкт реалізує інтерфейс, що має метод для отримання посилання на залежність.

Наведемо приклад використання ІЗ. Приведені приклади дуже обмежені, але достатні для того, щоб стисло проілюструвати використання ін'єкції залежностей.

Розглянемо клас, потрібний для представлення автомобіля. Характеристики автомобіля, такі як вага, розміри, тип двигуна отримуються із зовнішнього джерела і присвоюються об'єкту. Класи, що читають характеристики, всі реалізують інтерфейс DataReader. Об'єкт автомобіль може виглядати так (приведені лише релевантні частини коду):

```
public class Car {
        private DataReader reader;
        ... // інші параметри, що належать
автомобілю
        public Car() {
                DataReader reader = new
        DataFileReader("cardata.txt");
        }
}
```

Неважко помітити, що об'єкт автомобіль створює свою залежність –DataReader – явно, через оператор «new». Більш гнучким підходом було б використання паттерна *фабрика*.

Змінимо клас автомобіль, щоб він використовував ІЗ через конструктор. Клас тепер виглядає так:

```
public class Car {
        private DataReader reader;
        ... // інші параметри, що належать
автомобілю

        public Car(DataReader reader) {
this.reader = reader;
        }
}
```

Значуща різниця у цьому варіанті класу полягає у наданні залежності класу автомобіль. Клас сам не знає, і не повинен знати, який саме тип зчитувача даних(data reader) він отримує.

Якщо ж використовувати відкладену ІЗ, то зчитувач даних (data reader) буде надаватися через спеціальний метод, вже після створення об'єкта.

```
public class Car {
        private DataReader reader;
        ... // other variables representing the
car

        public Car() { }

        public void setReader(DataReader
reader) {
        this.reader = reader;
        }
}
```

При використанні ІЗ жоден з класів (Car чи DataReader) не зобов'язаний знати про існування іншого чи явно викликати один одного. Клас автомобіль (Car) просто приймає об'єкт зчитувача даних і не пот-

ребує додаткової логіки створення об'єкта зчитувача даних. Це дозволяє класу автомобіль зосередитися на значущій «бізнес логіці», а не створювати громіздку і негнучку інфраструктуру отримання та інтегрування залежностей [8].

Приведені приклади не пояснюють, як ІЗ налаштовується. Це залежить від використовуваного каркасу побудови ІЗ та може бути зроблено за використання конфігураційних файлів, анотацій або програмно.

Далі розглянемо приклад використання каркасу ІЗ в SF.

В цьому випадку, для попереднього прикладу слід налаштувати зв'язки об'єктів. У каркасі SF найбільш поширений спосіб конфігурування зв'язків – це використання xml-файлів [9]. Починаючи з версії 2.5, також стало можливо використання анотації для конфігурування ІЗ.

У нашому прикладі класи DataReader та Car потребують конфігурування. Наведемо приклад частини XML-конфігурації:

```
<bean id="filereader" class="
com.masterproject.di.DataFileReader">
    <constructor-arg
value="cardata.txt"/>
</bean>
<bean id="car"
class="com.masterproject.di.Car">
    <property name="reader"
ref="filereader"/>
</bean>
```

Атрибут «Id» використовується для ідентифікації об'єкта, або як його ще часто називають в термінології SF – bean. Атрибут «class» показує, якого класу слід створити об'єкт. У нашому прикладі екземпляр класу DataFileReader створюється з використанням ін'єкції через конструктор для вказування файлу з даними.

Об'єкт Car використовує відкладену ІЗ для автоматичного отримання посилання на зчитувач даних (file reader) після власного створення. Це досягається завдяки використанню посилання на об'єкт з «id» filereader.

Наступний фрагмент коду показує як отримати посилання на екземпляр класу Car використовуючи BeanFactory.

```
package com.masterproject.di;

import
org.springframework.beans.factory.BeanFactory;
import
org.springframework.beans.factory.xml.XmlBeanFactory;
import
org.springframework.code.io.FileSystemResources;

public class Example {
    public static void
main(String[] args) throws Exception
{
        BeanFactory factory =
        new
    XmlBeanFactory(new
    FileSystemResource ("config.xml"));
        Car car = (Car)
    factory.getBean("car");
        car.doStuff();
    }
}
```

Використання getBean() методу BeanFactory об'єкта з попередньо визначеним аргументом id, призводить до отримання посилання на екземпляр класу Car.

Коли об'єкт визначено, межі за замовчуванням встановлюються в позицію – singleton. Це означає існування лише одного спільного екземпляра класу для всіх наступних викликів і посилань. Протилежним цьому є значення межі – prototype, за якою новий об'єкт класу буде створюватися кожного разу коли посилання буде отримано. Також існують додаткові типи меж. Наприклад, можливим є налаштувати об'єкт на існування протягом життєвого циклу HTTP запиту або сесії.

Spring-контейнер також підтримує автоматичне зв'язування визначених об'єктів. Використання автоматичного зв'язування дозволяє вирішити, як об'єкти мають бути пов'язані шляхом вивчення

самих класів, використовуючи імена або типи атрибутів. За допомогою нього досягається відчутне зменшення або навіть відмова від зв'язування, основаного на XML конфігуруванні. Недоліком такого підходу до зв'язування об'єктів є відсутність наглядної картини зв'язків між об'єктами (не існує явної конфігурації).

Програмний каркас SF дозволяє поєднати автоматичне та явне зв'язування об'єктів.

Можна використовувати й інші популярні Java каркаси ін'єкцій залежностей. Наприклад, Guice – каркас ін'єкції залежностей з відкритим кодом для Java 5, розроблений Google. Він явно використовує анотації для зв'язування залежностей. Програмний каркас PicoContainer замість анотацій чи XML-файлів надає спеціалізований API для програмної конфігурації об'єктів. EJB 3 підтримує ін'єкцію залежностей двох типів; ін'єкцію атрибутів класу та ін'єкцію через спеціальний метод (setter injection) [10]. При ін'єкції атрибутів класу контейнер зберігає посилання на залежність у приватному атрибуті класу без використання якогось публічного методу.

Ін'єкція залежностей стала настільки невід'ємною частиною Spring Framework, що з достатньою впевненістю можна стверджувати, що без неї розробка в рамках даного каркасу майже не можлива, хіба, що лише при використанні окремих модулів Spring Framework. Всі класи в Spring Framework спроектовані так, що відбувається ін'єкція залежності, де це лише можливо.

Використання ін'єкції залежностей жодним чином не впливає на бізнес-логіку програмного застосування чи об'єктно-орієнтований дизайн класів. Натомість, воно відкриває альтернативний спосіб отримати посилання на залежні об'єкти. Саме через це, потрібні лише незначні, а часто і взагалі ніякі модифікації до загального дизайну системи для того, щоб перейти на використання ін'єкції залежностей. Якщо з певних причин каркас, що підтримує ін'єкцію залежностей не може бути використаний, то концепція і проектний взірець (паттерн) можуть бути з великою користю задіяні у проектуванні класів.

Реалізація класів з інтерфейсами, спроектованими для підтримки setter- і constructor-ін'єкцій та створення цих класів, використовуючи паттерн типу *фабрика*, принесе чимало користі, яку можна порівняти з користю від використання ін'єкції залежностей, але в цьому випадку немає потреби використовувати повноцінний каркас.

Загалом, при використанні ін'єкції залежностей зменшується кількість шаблонного програмного коду, адже зникає потреба у створенні і дублюванні коду, який здійснює пошук або створює залежні об'єкти. Це пояснюється тим, що залежності, як правило, передаються через setter-методи або в конструктор класу, на відміну від їх створення, ініціалізації та отримання через методи пошуку або використовуючи *фабрики*.

Менша кількість шаблонного програмного коду призводить до вищого ступеню зв'язності (cohesion) об'єктів, що, звісно, завжди є бажаним. В цьому випадку програмний код фокусується менше на питаннях інфраструктури, таких як створення і отримання залежних об'єктів, і більше на виконання задач застосування. Наприклад, можна встановити всі зовнішні залежності певного об'єкта лише подивившись на сигнатуру конструктора та публічних setter-методів.

Варто окремо відзначити потенційну небезпеку одночасного використання традиційного способу отримання залежностей і ін'єкції залежностей. У такому випадку дуже легко не помітити залежності, які можна ідентифікувати, лише уважно переглянувши реалізацію метода та лише проаналізувати сигнатури методів або конфігурацію ін'єкції залежностей.

Ін'єкція залежностей пропагує дизайн слабше поєднаних класів, адже їхні залежності отримуються ззовні, на противагу створенню або отриманню залежностей, використовуючи інфраструктурний програмний код, що знаходиться в самому об'єкті. Лише це само по собі робить можливу зміну класу легшою.

Прикладом, коли потрібна зміна програмного коду, може бути ситуація, коли змінюються залежності. Наприклад,

колекція даних змінюється з одного типу даних на інший. Зміни в класі потенційно простіші у системі, що побудована з використанням ін'єкції залежностей, у порівнянні з традиційним підходом до проектування. Особливо це проявляється, коли ін'єкція залежностей використовується разом з концепцією проектування за інтерфейсом.

Причиною простішої реалізації можливих змін в цьому випадку є передача залежності ззовні і єдиний «контракт» між залежністю і об'єктом, що реалізує логіку, та те, що використовує цю залежність лише інтерфейс. Якщо залежність змінюється, але все ще використовується попередній інтерфейс, то щодо об'єкта слід здійснювати незначні зміни або взагалі не проводити змін.

Очевидно, що модульне тестування також виграє від використання ін'єкції залежностей. Загалом менш зв'язані об'єкти легше тестувати модулями, але якщо присутні залежності, то вони мають бути замінені на об'єкти-макети.

Об'єкт-макет – це об'єкт, що симулює поведінку справжнього об'єкта, адже використання справжніх об'єктів непрактичне в модульних тестах. Існує декілька причин використання об'єктів-макетів. Прикладом можуть бути об'єкти зі специфічними станами, які дуже складно відтворити (наприклад, помилки мережі) або вони занадто повільні (певні запити до бази даних). У модульному тестуванні веб-застосувань загальноприйнятою практикою є використання об'єктів-макетів, що репрезентують HTTP запит.

Ін'єкція залежностей дозволяє легку заміну реальних об'єктів макетами, де це необхідно. Якщо б залежності були не так легко досяжні, то їх заміна була б складною операцією.

2. Аспектно-орієнтоване програмування

Наразі, об'єктно-орієнтоване програмування (ООП) є де-факто стандартом для більшості нових проектів розробки програмного забезпечення [10]. ООП започаткувало об'єктну абстракцію і полегшує моделювання загальної поведінки та інкапсулювання цієї поведінки модульним способом. Модульність робить систему гнучкішою, легшою у підтримці й швидшою у розробці [11]. На жаль, і ООП не забезпечує прийнятного дизайну і ефективної розробки наскрізної функціональності, тобто такої функціональності, що «прошиває» всю систему, як, наприклад, безпека [9].

Це часто призводить до компромісів, коли програмний код для певної задачі розсіюється по кількох модулях, що призводить до заплутаного програмного коду і одразу робить реалізацію як самої задачі, так й всіх модулів, що залежать від даної реалізації, складнішими у розробці та подальшій підтримці. Повторимо, що таку функціональність називають наскрізною (cross-cutting), оскільки вона перетинає різні модулі. Організацію реалізації такої функціональності прагне спростити АОП, шляхом надання способу відокремлення функціональності в модульні аспекти.

Для подальшого ознайомлення із аспектно-орієнтованим програмуванням, використаємо приклад, що спрощено описує систему адміністрування готелем. Система адміністрування використовується персоналом готелю для бронювання кімнат для гостей, резервування столиків у ресторані, і тому подібне.

На перший погляд модуль (або клас), відповідальний за бронювання номерів, здається досить тривіальним.

Спочатку перевіряється, чи є вільний номер на потрібний період. Якщо номер вільний протягом проміжку часу, що цікавить клієнта, то далі потрібно зарезервувати номер на вказаний термін. Варто зауважити, що не кожен з персоналу готелю повинен мати змогу здійснювати бронювання номерів. Тобто необхідна додаткова перевірка для гарантування, що поточний виконавець авторизований для здійснення даної процедури. Окрім, кожна операція має бути записана у спеціальний журнал та збережена в базі даних, щоб потім була можливість переглянути, хто, що і коли робив. Потрібно зберігати й іншу статистичну інформацію для менеджменту готелю.

Додавання викликів функцій перевірки прав користувача та здійснення облікового запису в журнал є відносно простою задачею, але варто розуміти важливий момент – додавання цих викликів породжує залежності на зовнішні модулі, чим перетворюють модуль резервування номерів жорстко залежним від інших частин системи. До того ж знижується зв'язність (внутрішній взаємозв'язок між частинами) модуля, адже система резервування номерів тепер мусить відповідати за автентифікацію користувача та ініціювати створення облікового запису про виконану дію.

Схема, показана на рис. 1, як модуль для резервування викликає методи модулів ведення обліку.

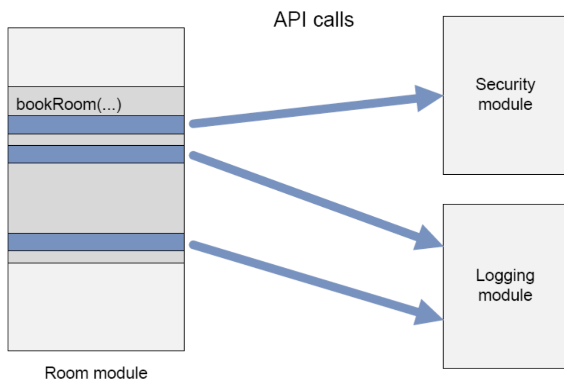

Рис. 1. Схема викликів

У аспектно-орієнтованому програмуванні таку наскрізну функціональність, як перевірка прав користувачів на виконання певних дій та здійснення записів до журналу обліку можна помістити у окремі модулі – аспекти. Аспект може налаштовуватися на автоматичне виконання, керуючись набором простих правил, що забирає відповідальність за виконання відповідних задач з окремого модуля, класу чи функції, до яких він буде застосований.

Наприклад, аспект перевірки прав користувача може бути автоматично виконаний перед кожним викликом функції з ім'ям, що починається зі слова "book" (наприклад, bookRoom(…)). Тобто, відповідна функція буде виконана лише за умови, що користувач має відповідні права на виконання.

Схема із рис. 2 демонструє процедуру резервування номерів із застосуванням аспектно-орієнтованого підходу. Введені аспекти визначають коли і як здійснювати виклики до API модулів обліку та безпеки.

Використання аспектно-орієнтованого програмування у даній системі призвело до зменшення розпорошення програмного коду та зробило його менш заплутаним. В цьому випадку модуль бронювання номерів лише містить програмний код, який має відношення лише до процесу бронювання. Це призвело до меншої взаємозалежності модулів, більш зрозумілого їх наповнення, а отже і покращило модульність системи.

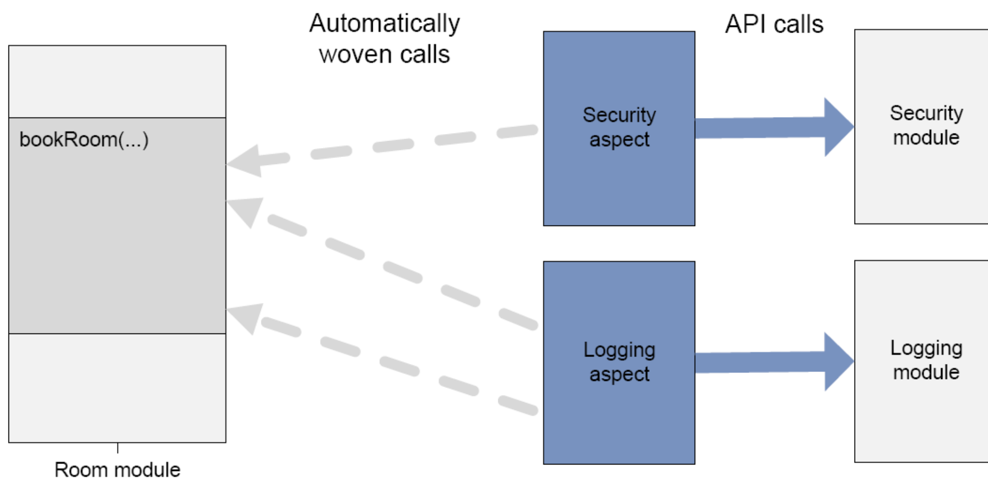

Рис. 2. Використання АОП для покращення ведення обліку та аутентифікації користувачів

Для докладнішого обговорення АОП, деякі терміни потребують детальнішого пояснення. Це терміни рекомендація (advice), точка з'єднання (join point), зріз точки (pointcut), аспект (aspect), ціль (target), зв'язування (weaving).

Рекомендація описує, що саме потрібно робити, коли застосовується аспект. Вона містить програмний код, який забезпечує виконання дій, що мають бути застосовані. Наприклад, якщо мета аспекта полягає у веденні журналу обліку подій, які виконуються у застосуванні, рекомендація буде містити програмний код, що призведе до запису відповідного рядка у файл облікових записів.

Рекомендація може бути виконана до або після точки з'єднання. Вона може замінити програмний код точок з'єднання, або навіть пропустити його виконання.

Точки з'єднання – це всі можливі місця виконання рекомендації у програмному коді. Вони називаються точками з'єднання, оскільки представляють місця в програмному коді, де рекомендації (advices) можуть бути під'єднанні (або підключені).

Можливими точками під'єднання можуть бути виклики методів, створення об'єктів або місця зміни атрибутів об'єктів. Різні типи точок з'єднання, що підтримуються, варіюються залежно від використання різних аспектно-орієнтованих програмних каркасів.

У прикладі ведення журналу обліку, точками з'єднання будуть всі можливі місця у програмному коді, де може бути викликана рекомендація ведення обліку.

Зріз точки визначає, де рекомендація має бути виконана, вказуючи набір точок з'єднання. Як правило, зрізи точок вказують точки з'єднання шляхом переліку відповідних назв класів або процедур явно, або використовуючи шаблони регулярних виразів.

Різні варіанти зрізів точок варіюються між різними імплементаціями програмних каркасів [12]. Деякі з них надають можливість створення динамічних зрізів точок, які залежать від рішень прийнятих під час виконання, наприклад, застосування рекомендації у певній ситуації може залежати від значення певного параметру.

Продовжуючи приклад з журналом обліку подій, зріз точки вирішує, коли має бути створений обліковий запис (з-поміж усіх можливих точок з'єднання). Наприклад, це може відбуватися, коли викликається будь-яка процедура з іменем, що починається зі слова "book".

Найчастіше вживаними типами зрізів точок є "до", "після" і "навколо". Перші два викликаються за мить до і після виклику певної процедури, а рекомендація "навколо" інкапсулює виклик процедури і, таким чином, надає можливість виконати програмний код до і після, включаючи можливість пропуску виклику процедури взагалі.

Центральний елемент, у якому рекомендація і точки з'єднання поєднуються, називається аспектом. Окрім перелічених аспектно-орієнтованих специфічних деталей, аспект може містити змінні, процедури і внутрішні класи, точно так як і звичайні Java класи.

Аспект у прикладі про журнал обліку подій визначає все довкілля процесу ведення обліку: де облік має бути застосованим (зріз точки), коли і як запис обліку має бути здійснений (рекомендація).

Ціль – це об'єкт, до якого аспект має бути застосований. Зв'язування представляє процес поєднання аспектів і об'єктів-цілей у завершену систему. Аспекти «сплітаються» у об'єкти у вказаних точках з'єднання. Зв'язування може бути виконано у декілька способів: під час компіляції чи під час виконання.

Для уточнення АОП розглянемо навмисно спрощений приклад, що не містить складної бізнес-логіки, а це дозволить з легкістю простежити потік виконання. Наша головна мета – показати відмінність АОП від «чистого» ООП.

Головний клас прикладу – Room містить метод bookRoom(int i), що приймає на вхід ціле число (номер кімнати) і бронює відповідну кімнату у готелі. Головний метод просто робить два виклики для демонстрації потоку виконання під час виконання bookRoom() метода.

```
public class Room {
	public void bookRoom(int id) {
		System.out.println("Roo
	m " + id + " booking logic
	performed");
	}

	public static void
main(String[] args) {
		Room room = new
	Room();
		room.bookRoom(1);
		room.bookRoom(0);
	}
}
```

Далі маємо Logging аспект, що здійснює збереження облікового запису до і після виклику метода bookRoom(int i) класу Room. Зріз точки ведення обліку визначає, що створення запису буде відбуватися до і після виконання усіх публічних методів класу Room, назви яких починаються з "book", незалежно від значення, яке поверне метод, і параметрів, які він приймає. Дві точки з'єднання визначено для виконання до і після того, як викликається зріз точки створення облікового запису.

```
public aspect Logging {
	public pointcut logging():
call(public* com.Room.book*(*));
	before(): logging() {
		System.out.println(
		"Entering: " +
	thisJoinPointStaticPart.getSign
	ature().getName()
		);
	}
	after(): logging() {
		System.out.println(
		"Exiting: "	+
		thisJoinPointStaicPart.
		getSignature().getName
		()
		);
	}
}
```

Аспект перевірки прав користувача на виконання певної операції має схожий зріз точки, але він застосовує рекомендацію «навколо» і лише умовно продовжує виконання початкового виклику. Дуже спрощений програмний код у прикладі міг би бути замінений на логіку, що перевіряє, чи має поточний користувач дозвіл на резервування певного номера. Якщо користувач дозволу не має, то виклик до метода, що здійснює резервування, не відбудеться (як показано у приклад, і коли параметр «і» дорівнює 0).

```
public aspect Security {
	public pointcut access():
call(public* com.Room.book*(int));

	void around(int i): access()
&& args(i) {
		if (i == 0) {
			System.out.print
		ln("Blocked call when i
		== " + i);
			return;
		} else {
			proceed(i);
		}
	}
}
```

Виконання цього тривіального прикладу за допомогою AspectJ згенерувало такий результат:

```
Entering: bookRoom
Room 1 booking logic performed
Exiting: bookRoom
Entering: bookRoom
Blocked call when i == 0
Exiting: bookRoom
```

Для кожного виклику створюється запис у журнал обліку подій, варто звернути увагу, що для кожного виклику аспектом створюються два записи Logging – до і після виконання. У випадку, коли викликається метод bookRoom(int i) з аргументом i=0, він блокується аспектом перевірки прав користувачів.

Архітектура. Порівняно з ін'єкцією залежностей АОП має великий вплив на проектування та розробку застосування. З концепцією поділу розрізненої функціона-

льності на аспекти архітектура застосування з використанням АОП значно відрізняється від використання лише загальноприйнятого ООП.

При успішному використанні АОП, наскрізна функціональність, яка раніше була розсіяна по всій системі, у значній мірі може бути зосереджена в аспекті та буде застосована там, де це необхідно. Це призводить до двох важливих результатів. Програмний код, що викликає наскрізну функціональність, може бути видалений з об'єктів. У цьому випадку можна сфокусувати об'єкти лише на логіці застосування і таким чином досягти вищої зв'язності (cohesion) кожного окремого об'єкта. З іншого боку, реалізація і запуск наскрізної функціональності зосереджується у аспектах, а отже у випадку необхідності внесення змін до програмного коду наскрізної функціональності, потрібно лише змінити відповідний аспект, а не всі (у найгіршому випадку) місця у системі, де ця функціональність використовувалася. Ще одним типовим прикладом наскрізної функціональності є ведення обліку подій у системі.

АОП привносить новий підхід до проектування. Важливо пам'ятати, що АОП є доповненням до ООП. Розробник має колізію вибору – яка функціональність підходить для реалізації за допомогою аспекту, а яка ні.

Базуючись на досвіді проектування застосувань, ми пропонуємо використовувати два підходи до ідентифікації аспектів.

Перший підхід полягає у ідентифікації наскрізної функціональності під час аналізу вимог до застосування і початкового етапу проектування архітектури. Цей підхід включає ідентифікацію бізнес-аспектів, що пронизують різні модулі.

Другий підхід полягає у розробці модулів і об'єктів у традиційний спосіб з подальшою переробкою окремої функціональності на аспекти, коли стане зрозуміло, що ця функціональність є наскрізною, наприклад, якщо програмний код повторюється у багатьох місцях системи; об'єкти переоблюються в такий спосіб, що вони використовуватимуть аспект, а не повторюватимуть той самий програмний код.

Зазначимо, що доволі легко застосовувати аспекти для типових рішень, які описані у багатьох наукових статтях як приклади: безпека, облік і керування транзакціями.

При виборі використання АОП, як правило, обирається другий варіант.

Існує кілька причин такого вибору. По-перше, завжди легше робити щось «звичним способом» порівняно до впровадження чогось нового, та ще й за умови, що розробник не має значного досвіду роботи з цим новим. По-друге, більшість проблем проектування не нові й тому вже існують перевірені часом вирішення більшості з них.

Така ситуація часто виникає при використанні Spring Framework. Незважаючи на те, що Spring Framework має власну AOP підтримку, вже існує чимало придатних способів вирішення проблем без використанням АОП.

Аналогічне можна сказати і щодо використання паттернів проектування (design patterns). Як правило, краще використовувати перевірені паттерни замість впровадження аспектно-орієнтованого вирішення. Однак однозначної думки тут не існує. Наприклад, у роботі [7] досліджувалися 23 паттерни Gang-of-Four. Вони були реалізовані як з використанням об'єктно-орієнтованих, так і аспектних рішень. Після порівняння зроблено висновок, що більшість аспектно-орієнтованих рішень покращили поділ пов'язаної з паттернами функціональності. Та в праці [13] зазначають, що тільки чотири з 23 взірців мають значне повторне використання у проектах.

Потрібно брати до уваги і те, що хоча кращий поділ функціональності може бути досягнутий шляхом використання аспектно-орієнтованих рішень, у деяких випадках це може призвести до збільшення розміру програмного коду і складнішого дизайну системи. У таких випадках загальноприйнятий підхід може виявитися бажаним.

Зрозуміло, що АОП є відносно новим способом проектування застосувань у порівнянні до ООП і коли розробник стає досвідченішим у застосуванні аспектно-орієнтованих рішень, бажання натомість

застосовувати традиційні рішення через те, що вони більш вивчені, поступово зменшуватиметься.

Підтримка. З точки зору підтримки застосування, при використанні АОП фундаментальним постає питання: чи людина, що змінює програмний код (не автор програмного коду), має досвід і достатні знання з АОП. Може виявитися, що елементарного перегляду класів і аналізу програмного коду в кожному з методів стане недостатнім для розуміння заміни потоку виконання програмного коду аспектами. Намагання зрозуміти, чи варто відлагоджувати (debug) таку систему без достатнього розуміння того, які саме аспекти застосовуються, може виявитися проблематичним.

У такій ситуації важливу роль відіграють середовища програмної розробки (IDE), які підтримують аспектно-орієнтоване програмування. Вони суттєво спрощують намагання зрозуміти, де і як застосовуються аспекти. Навіть для спеціалістів зі значним досвідом у АОП вони надають значну допомогу при інтерпретації складних умовних зрізів (pointcuts).

Як ми вже зазначали раніше, при використанні АОП, програмний код, що реалізує логіку наскрізної функціональності, можна видалити з усіх методів і натомість розташувати в окремий модуль – аспект. Це призводить до кращої зв’язності класів і робить їх можливі подальші зміни легшими. Полегкість зумовлюють дві причини: програмного коду у кожному з класів стає менше і він стає більш сфокусованим, а відповідальність за виклик логіки наскрізної функціональності перенесена з методів класу до аспектів.

Однією з найбільших переваг аспектно-орієнтованого підходу є значне полегшення змін програмного коду наскрізної функціональності. Якщо взяти за приклад ведення журналу обліку подій системи, то виклики методів створення записів можуть бути розкидані за багатьма класами. І якщо виникає потреба привнесення змін до програмного коду створення записів, то у найгіршому випадку буде необхідна зміна програмного коду, що викликав методи створення записів, за всією системою. Зрозуміло, що це збільшує ймовірність появи помилок і просто вимагає чимало часу та зусиль при умові наявності великої кількості програмного коду в системі.

При використанні АОП такі зміни ізолюються у аспекті, що відповідає за внесення записів до журналу і таким чином поєднує у собі як логіку зберігання, так і визначення способу та часу його виклику зрізами (pointcuts).

Варто зазначити, що існує ризик неправильності призначення аспекту при зміні класів, до яких він застосовується. Зрізи можуть бути визначені за допомогою назв методів і тому при зміні класу розробник має дотримуватися домовленості щодо назв методів. Тому бачиться абсолютно необхідним для розробника, що змінює програмний код, знати про використані аспекти, навіть, коли зміни жодним чином не стосуються наскрізної функціональності. Ігнорування цього може призвести до помилок виконання програмного коду, які буде надзвичайно складно знайти.

Тестування. У традиційній архітектурі наскрізну функціональність неможливо тестувати з модульних тестів (unit test), бо ця функціональність розсіяна, і тому не є модулем. При аспектно-орієнтованій архітектурі наскрізна функціональність зібрана у аспекти, тобто ми маємо окремі модулі.

Варто зазначити, що модульне тестування аспектів не таке просте, як тестування простих класів, адже, логіка, яку виконує аспект, завжди застосовується до цільового об’єкта. Під час розробки застосування були ідентифіковані три різні підходи до тестування.

Найпростішим способом такого тестування вважають модульний тест об’єктів, до яких застосовані аспекти. В цьому випадку ми певною мірою отримуємо інтеграційний тест, адже одночасно тестуються не аспектно-орієнтовані класи й аспекти, що застосовуються до цих класів.

Написання такого модульного тесту нескладне, адже він за своєю природою нічим не відрізняється він інших модульних тестів. Варто відзначити, що додаткова увага має бути приділена перевірці ре-

зультатів виконання тесту. Він повинен мати негативний результат як при умові неправильного виконання логіки класу, так і за умови, що аспект працює неправильно.

Недоліком інтеграційного тестування вважають неізольоване тестування аспекту. Тому, якщо тестування цільового об'єкта дає негативний результат, то і тестування аспекту також поверне негативний результат, навіть за умови, що аспект відпрацьовує коректно. Можливим рішенням бачиться тут використання макету цільового об'єкта, до якого застосовується аспект. З точки зору аспекту, такий макет нічим не відрізняється від інших цільових об'єктів.

Створення макету цільового об'єкта має на меті звузити наповнення макета лише до програмного коду, що допомагає перевірити потік виконання і коректний виклик рекомендацій у правильних точках з'єднання. Тобто макет не повинен містити іншої логіки, подібної до тієї, що містять звичайні цільові об'єкти.

Перевагою використання макета цільового об'єкта є те, що логіка аспекту може бути протестована ізольовано. Недоліком можна назвати, те, що потенційно має бути створена чимала кількість макетів цільових об'єктів, якщо аспекти застосовуються до багатьох об'єктів.

Третій підхід полягає у делегуванні логіки аспекту в окремі класи і вже потім виконувати модульне тестування цих класів. Цей підхід перевіряє логіку, що виконується всередині аспекту, але не може перевірити, коли рекомендації викликаються визначеннями точок зрізів. Тому цей підхід може дуже вдало поєднуватися з використанням макетів цільових об'єктів. Недоліком підходу вважається незручність або складність виокремлення логіки в окремі класи.

3. Каркаси аспектно-орієнтованого програмування

У залежності від використовуваного програмного каркаса, що підтримує аспектно-орієнтоване програмування, набір підтримуваних можливостей може різнитися. Ця різниця включає у себе, те, які типи точок з'єднання можуть бути застосовані, які зрізи точок визначені і, як та коли відбувається зв'язування. Вказані відмінності розглянемо на прикладі найбільш використовуваних каркасів Spring [9], AspectJ [14], EJB 3 (Enterprise JavaBeans) [10].

Каркас *Spring AOP* використовується для підтримки аспектно-орієнтованого програмування у SF. Він дозволяє розробникам створювати власні аспекти простим і легким способом та надає декларативні комерційні сервіси (головним чином керування транзакціями) у самому каркасі [9]. Spring AOP не являється, і не планувався бути таким, повномасштабним AOP каркасом як AspectJ.

Рекомендація (advice) у Spring має синтаксис стандартного Java класу, а зрізи точок описані в XML конфігураційних файлах або з допомогою анотацій. Зв'язування у Spring відбувається під час виконання шляхом загортання об'єкта-цілі у проксі клас. Коли відбувається виклик проксі клас позиціонується як ціль і перехоплює виклики, де рекомендація має бути застосована. Через це відпадає потреба у спеціальному процесі компіляції, все є чистою Java і застосування може бути напряму використано у J2EE веб-контейнері або сервері застосувань.

Найпомітніший недолік Spring AOP – це підтримка лише точок з'єднання методів. Доступні типи рекомендацій:

• Before – виконується перед методом, але не може запобігти виконанню методу, хіба що у випадку виникнення помилки;

• After return – виконується після нормального виконання метода;

• After throwing – виконується якщо виконання метода завершилося помилкою;

• After (finally) – завжди виконується після виконання метода;

• Around – огортає точку з'єднання і може показувати різну поведінку до і після запуску метода. Можливий варіант, коли рекомендація вирішить пропустити виконання метода взагалі.

Якщо потрібні детальніші рекомендації, ніж ті, що надаються точками з'єднання методів, то також існує можливість використовувати AspectJ в межах Spring Framework.

AspectJ – найпоширеніший стандарт для аспектно-орієнтованого програмування [15]. Перша загальнодоступна версія AspectJ була презентована в 2001. Нині AspectJ – проект з відкритим програмним кодом у Eclipse Foundation [3]. AspectJ – це розширення Java для опису зрізів точок і способів їх використання. Java використовується для розробки внутрішніх concerns.

Будь-яка Java програма є прийнятною AspectJ програмою і AspectJ компілятор генерує стандартні Java файли класів, що можуть бути виконані будь-якою Java віртуальною машиною. AspectJ підтримує compile-time зв'язування, post-compile зв'язування (зв'язування аспектів у вже зкомпільовані класи) і load-time зв'язування (LTW). При load-time зв'язуванні Java віртуальна машина використовує завантажувач класів.

EJB 3 (Enterprise JavaBeans), частина JavaEE 5, підтримує аспектно-орієнтоване програмування з точки зору використання перехоплювачів (interceptors). Підтримка AOP обмежена [10].

Перехоплювачі – це об'єкти, які автоматично виконуються, коли метод викликається у EJB об'єкті. Типи точок зрізів, що підтримуються: around рекомендація, яка дозволяє налаштування гнучкої поведінки як на початку метода, так і після нього, з можливістю реакції як на результат що повернув метод, так і на можливу помилку, до якої призвело виконання методу.

Висновки

У роботі проаналізовано використання ін'єкції залежностей і АОП при розробці складних програмних проектів.

Ін'єкція залежностей легка у використанні та розумінні і не вимагає суттєвих змін архітектури класів певного застосування. Однак вона привносить паттерн проектування, що призводить до менш залежних модулів і меншої кількості шаблонного, повторюваного програмного коду.

Особливо вигідним є використання ін'єкції залежностей у тестуванні, особливо модульному. Зрозуміло, що модульне тестування полегшується, якщо клас не відповідає за пошук або створення своїх залежностей і ресурсів. Частково тому, що класи стають менш зв'язаними між собою, але більшою мірою тому, що залежності легко можна підмінити об'єктами-макетами.

Чимало програмних каркасів надають підтримку ін'єкції залежностей. Саме тому використання ін'єкції залежностей досить легке і зрозуміле. Якщо з певних причин використання програмного каркаса, що підтримує ін'єкцію залежностей неможливе, може бути отримана певна користь від використання принципу інверсії контролю.

Основна мета АОП полягає у розробці і підтримці наскрізної функціональності у більш модульний спосіб порівняно з традиційним об'єктно-орієнтованим програмуванням.

Це досягається впровадженням концепції аспектів, що застосовуються до цільових об'єктів. Опис точки зрізу використовується для визначення того коли, де і які зміни параметрів чи виклики методів цільових об'єктів перехоплюються програмним кодом з аспекту. Визначення відбувається через процес, що має назву зв'язування (weaving). У випадку AspectJ зв'язування як правило, виконується з використанням AspectJ компілятора. У випадку Spring AOP (підтримка АОП у Spring Framework) зв'язування відбувається під час виконання і тому не вимагає спеціального компілятора, тим самим дозволяючи легкий перехід на використання АОП.

Щоб отримати відчутну перевагу від використання АОП варто перейти на дещо новий підхід до процесу проектування та розробки у порівнянні до традиційного ООП, для того, щоб ідентифікувати функціональність, яка претендує на місце в аспекті.

Незважаючи на існування чималої кількості ситуацій, коли виграш від використання АОП беззаперечний, воно привносить певну складність, адже потік виконання програмного коду цільового об'єкта може бути змінено і це буде непомітно з програмного коду. Використання аспектів також вимагає зміни до процесу тестування, особливо модульного тестування.

1. *Андон П., Дерецький В.* Проблеми побудови сервіс-орієнтованих прикладних інформаційних систем в semantic web середовищі на основі агентного підходу // Проблеми програмування — 2006. – № 2 – 3 [спец. вип.]. – С. 493–502.
2. *Андон Ф.И., Бабко Л.Д.* Стандартизация инженерии систем и программных средств в Украине // Кибернетика и системный анализ. – 2009. – 45, № 6. – С. 144–148.
3. *Aspectj* проект Eclipse Foundation. http://www.eclipse.org/aspectj/.
4. *Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu* Professional Java Development with the Spring Framework, Wiley Publishing Inc, July 8, 2005. – P. 1–6.
5. *Ralph E. Johnson and Brian Foote.* Designing reusable classes // Journal of Object-Oriented Programming. – 1988. – 1(2). – P. 22–35.
6. *Martin Fowler.* Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html , 2004.
7. *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.* Design Patterns. Addison-Wesley Professional. – 1995. – P. 87–97.
8. *Dhanji R. Prasanna.* Dependency Injection. Manning Publications Co., Greenwich, CT, USA, 2008. – P. 19–20.
9. *Rod Johnson.* The Spring Framework – Reference Documentation. http://static.springframework.org/spring/docs/2.5.x/reference/.
10. *Debu Panda, Reza Rahman, and Derek Lane.* Ejb 3 in Action. Manning Publications Co., Грінвіч, CT, США. – 2007. – P. 146–150.
11. Parnas D.L. On the criteria to be used in decomposing systems into modules. – 1979. – P. 139–150.
12. *Maximilian Storzer and Stefan Hanenberg.* A classification of pointcut language constructs. In Software-engineering Properties of Languages and Aspect Technologies. 2005.
13. *Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa.* Modularizing design patterns with aspects: a quantitative study // In AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, 2005. – P 3–14.
14. *Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold.* An overview of aspectj. In ECOOP, Springer, 2001. – 2072, P. 327–353.
15. *Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin.* Aspect-oriented programming // In Proceedings European Conference on Object-Oriented Programming, Springer-Verlag, Берлін, 1997. – 1241. – P. 220–242.

Одержано 18.10.2012

Про авторів:

Глибовець Микола Миколайович,
доктор фізико-математичних наук,
професор,
декан факультету інформатики,

Гороховський Семен Самійлович,
кандидат фізико-математичних наук,
доцент кафедри інформатики,

Луценко Ігор Миколайович,
магістр комп'ютерних наук,
керівник команди розробників
програмного забезпечення.

Місце роботи авторів:

Національний університет
"Києво-Могилянська Академія",
напрям наукових інтересів – розподілені
системи інтелектуального типу,
програмні системи підтримки
електронного навчання
254070, Київ-70,
вул. Сковороди, 2.
Тел. (067) 209 0740,факс (044) 416 4515,
E-mail: glib@ukma.kiev.ua
gor@ukma.kiev.ua

м. Київ, вул. Академіка Грекова 5\68,
фірма Otto (GmbH & Co KG).
Тел.: (050) 135 2319,
E-mail: lutsenko.i@gmail.com