***Досліджена проблема складності розробки та підтримки наскрізного функціоналу програмного забезпечення та її рішення за допомогою аспектно-орієнтованого підходу. Описано складність застосування аспектно-орієнтованого програмування в об'єктно-орієнтованих мовах програмування. Запропоновано архітектуру, яка забезпечить незалежність синтаксису оголошення та впровадження аспектів в об'єктно-орієнтовані програми***

***Ключові слова: аспектно-орієнтоване програмування, АОП, інтеграція аспектів, наскрізний функціонал, архітектура ПЗ, точка з'єднання***

***Исследована проблема сложности разработки и поддержки сквозного функционала программного обеспечения и её решения с помощью аспектно-ориентированного подхода. Описаны сложность применения аспектно-ориентированного программирования в объектно-ориентированных языках программирования. Предложена архитектура, которая обеспечит независимость синтаксиса определения и внедрения аспектов в объектно-ориентированные программы***

***Ключевые слова: аспектно-ориентированное программирование, АОП, интеграция аспектов, сквозной функционал, архитектура ПЗ, точка соединения***

УДК: 004.4
DOI: 10.15587/1729-4061.2016.63717

# ІНТЕГРАЦІЯ ЗАСОБІВ АСПЕКТО-ОРІЄНТОВАНОГО ПІДХОДУ У ОБ'ЄКТНО-ОРІЄНТОВАНУ МОВУ ПРОГРАМУВАННЯ

**В. М. Медведєва**
Кандидат технічних наук, доцент*
E-mail: bohdan.hukivskyi@gmail.com
**Б. М. Гуківський***
E-mail: bohdan.hukivskyi@gmail.com
*Кафедра автоматизації проектування енергетичних процесів та систем
Національний технічний університет України «Київський політехнічний інститут»
пр. Перемоги, 37, м. Київ, Україна, 03056

## 1. Вступ

Технології створення програм пройшли насичений шлях розвитку, було створено ряд мов програмування, алгоритмів, технологій і т. п. Існують різні підходи для написання програм. Кожен з цих підходів має свої переваги та недоліки, які визнають задачі, де може застосовуватися той чи інший підхід. При виборі підходу важливо враховувати як предметну область вирішуваної задачі, так і такі фактори і потреби, як швидкість написання рішення, відмовостійкість, надійність, стабільність, економічність та ряд інших факторів. Поєднавши найбільш успішні підходи, було створено ряд парадигм програмування.

Парадигма програмування – це набір ідей та понять, що визначають стиль написання програм [1]. Найбільш поширеними є: імперативне програмування, декларативне програмування, структурне програмування, функціональне програмування, логічне програмування та об'єктно-орієнтоване програмування. Кожна з цих парадигм знаходить застосування в різних областях програмування, але однозначно, на сьогодні найбільш розповсюдженим є об'єктно-орієнтоване програмування (ООП) [2]. Причина його успіху полягає у тому, що саме ООП вирішує одну з найзначніших проблем програмування – складність. Чим більша і складніша програма, тим важче розбити її на невеликі та чітко обмежені блоки. Розбиття програми на класи, які з певною мірою абстракції представляють реальні об'єкти, справляється з цією проблемою [2].

Але, хоч об'єктно-орієнтоване програмування представляє ряд способів для розподілення функціональності на модулі, функції та класи, існує функціонал, який не вдається винести в окрему сутність засобами ООП. Такий функціонал називають наскрізним, так як його реалізація розподілена по різним модулям програми [3]. Це призводить до розосередження коду програми, підвищує складність його розуміння та ускладнює супровід програмного забезпечення.

Ця проблема є надзвичайно актуальною, у зв'язку з необхідністю автоматизації великої кількості бізнес процесів. При автоматизації існує необхідність вирішити дві групи завдань, а саме: завдання, що пов'язані з логікою бізнес процесу, та завдання, пов'язані з сервісним функціоналом, велика частина якого є наскрізним. При високій складності систем рішенням цих завдань можуть займатися різні розробники, тому вкрай важливо мати можливість виділити їх в окремі модулі. Для вирішення цієї проблеми було створено аспектно-орієнтовану парадигму програмування.

Аспектно-орієнтований підхід (АОП) дозволяє виділити наскрізний функціонал в окрему сутність, що дозволяє розділити бізнес логіку та сервісний код. Основними поняттями АОП являються [4]:

– аспект – модуль чи клас, що реалізує наскрізний функціонал. Аспект змінює поведінку коду, застосовуючи пораду в точках з'єднання, що визначені певним зрізом;

– порада – засіб оформлення коду, який повинен бути викликаний із точки з'єднання;

– точка з'єднання – точка в програмі, де потрібно використати пораду;

– зріз – набір точок з'єднання;

– впровадження – зміна структури класу чи зміна ієрархії наслідування для додавання функціональності в код.

Проте існує проблема складності реалізації впровадження в об'єктно-орієнтованих мовах програмування.

## 2. Аналіз літературних даних і постановка проблеми

Об'єктно-орієнтовані мови програмування з легкістю дозволяють реалізувати такі сутності АОП, як аспект, порада, точка з'єднання та зріз, але не представляють стандартних засобів реалізації впровадження. Для його реалізації існує 2 основні підходи [1]:

1. Впровадження на етапі компіляції.

2. Впровадження на етапі виконання.

Впровадження на етапі компіляції має такі переваги, як висока швидкодія та можливість впровадження в будь-яких місцях програми [5]. Але має і суттєвий недолік – необхідність модифікації компілятора. Це призводить до ряду проблем. Так, програму, написану з використанням впровадження аспектів на етапі компіляції, неможливо компілювати на звичайному, не модифікованому компіляторі [2]. Це значно ускладнює процес командної розробки і розробки програм з відкритим сирцевим кодом. Іншим недоліком є те, що при оновлені версії компілятора, необхідно вносити відповідні зміни і в версію з підтримкою впровадження аспектів. Для мови програмування C# такий підхід використовує фреймворк PostSharp. Цей фреймворк модифікує стандартний процес компіляції, що дозволяє використовувати широкий спектр точок впровадження [6]. Але програма, написана за допомогою цього фреймворку, не може бути скомпільована стандартним компілятором. Для мови програмування Java існує фреймворк AspectJ з аналогічними перевагами та проблемами [4].

Впровадження на етапі виконання значно поступається в швидкодії та в гнучкості [7]. Більшість АОП фреймворків, що працюють на етапі виконання, використовують генерацію проксі класів, що, як мінімум, подвоює кількість викликів функцій під час виконання програми. Впровадження методом генерації проксі значно обмежує спектр можливих точок з'єднання. Але найбільшою проблемою є те, що неявна генерація проксі можлива лише за умови створення екземплярів об'єктів за допомогою контейнера залежностей, тобто лише в програмах, які використовують принцип інверсії залежностей [8]. В протилежному випадку, в програмі замість використання стандартного оператора створення об'єкту виникає необхідність використовувати фабрику – породжуючий шаблон проектування [3], що створить потрібний проксі. Незважаючи на численні недоліки, відсутність необхідності модифікації компілятора робить метод впровадження на етапі компіляції досить популярним, особливо враховуючи те, що більшість програм рівня корпорації використовують принцип інверсії залежностей [2]. Зазвичай АОП фреймворки такого типу являються модулями для конкретних Inversion of control (IOC) контейнерів, що унеможливлює зміну контейнера, без зміни аспектів. Такий підхід широко застосовується в мові програмування C#, наприклад, IOC контейнер Unity [9]. Для контейнера Castle Windsor існують модулі для підтримки АОП, а також є можливість самостійної реалізації АОП [10].

Оскільки обидва методи мають як переваги так і недоліки, існують бібліотеки для впровадження АОП як на етапі компіляції, так і на етапі виконання [11]. Але виникає проблема неможливості зміни типу впровадження без зміни коду. Адже бібліотеки мають різний синтаксис оголошення аспектів і ряд інших відмінностей. Крім того, розробник має вміти працювати з різними бібліотеками впровадження АОП, адже різні задачі можуть потребувати різні типи впровадження. Тому постає задача розробки фреймворка, який підтримуватиме різні типи впровадження аспектів з єдиним синтаксисом їх оголошення. Такий фреймворк повинен представити зручний синтаксис оголошення аспектів та систему, яка дозволить вибрати спосіб впровадження, або ж створити свій власний. Серед необхідних способів впровадження необхідно мати як способи на етапі компіляції, так і на етапі виконання. Крім того, способи інтеграції на етапі виконання повинні підтримувати DI фреймворки, а також надати спосіб інтеграції без використання DI.

## 3. Мета і завдання дослідження

Метою роботи є розробка фреймворку, який дозволить виконати впровадження наскрізного функціоналу в об'єктно-орієнтований програмний код. Важливою вимогою є незалежність оголошення та впровадження аспектів, що дозволить застосовувати розроблений фреймворк для вирішення різних типів задач.

Для досягнення поставленої мети були поставлені наступні завдання:

– розробити архітектуру, яка забезпечить незалежність впровадження від оголошення;

– розробити синтаксис оголошення аспектів та забезпечити можливість його розширення та модифікації;

– розробити засоби впровадження на етапі компіляції та розробити модифікацію компілятора для їх підтримки;

– розробити засоби інтеграції на етапі виконання та створити модулі для найбільш популярних IOC контейнерів для спрощення їх підключення;

– розробити прототип системи та модульні тести для перевірки якості та швидкості роботи.

## 4. Архітектура рішення проблеми

Для забезпечення незалежності опису та впровадження наскрізного функціоналу пропонується використати шаблон проектування міст. У такому випадку абстракцією виступатиме абстрактний клас Aspect, що включає в себе метод визначення зрізу та метод-пораду. А реалізацією виступатиме деякий абстрактний клас Introduction, з методом Introduce, який забезпечить інтеграцію аспекту. Різні реалізації цього класу можуть забезпечувати як впровадження на етапі компіляції так і на етапі виконання (рис. 1).

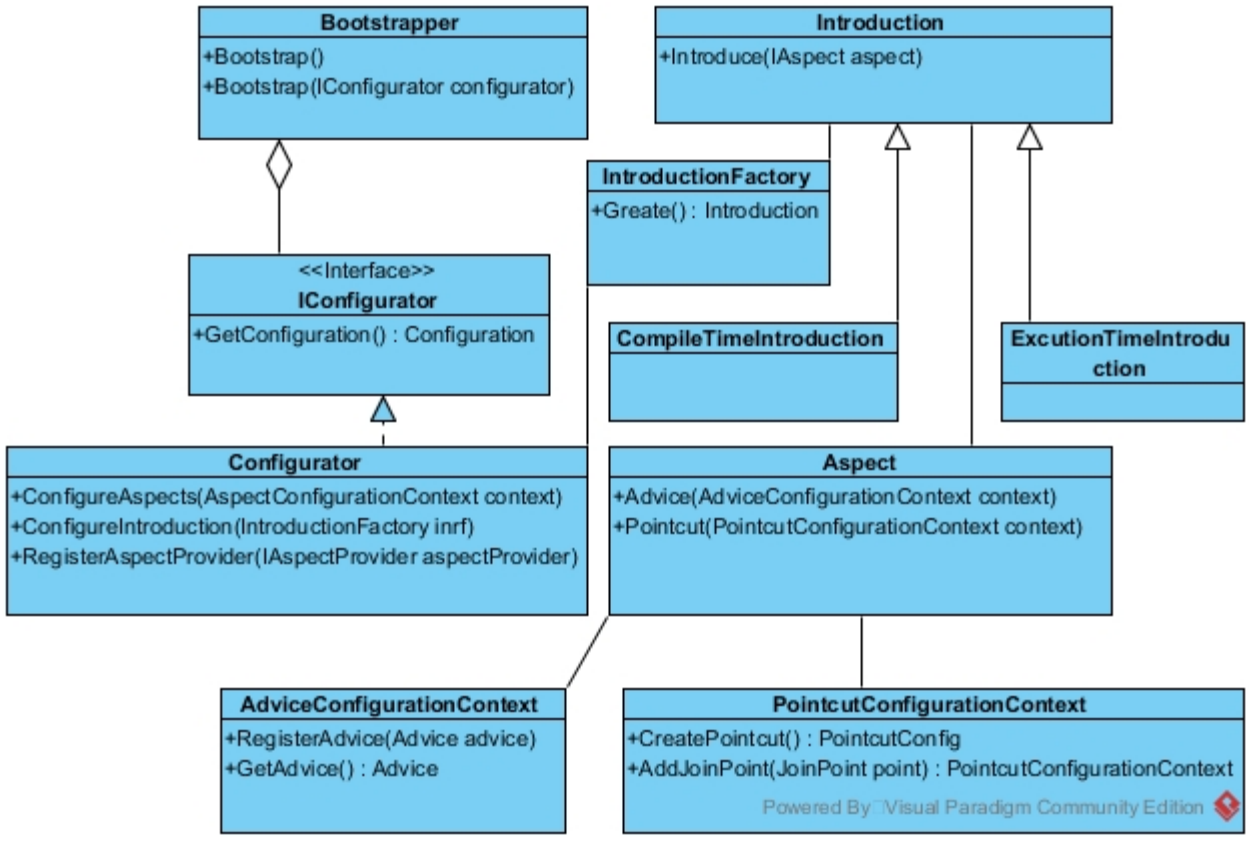


Рис. 1. Концептуальна діаграма класів

Клас Aspect може мати нащадків, які призначені для реалізації типового наскрізного функціоналу. Наприклад, клас LogAspect забезпечує ведення протоколу роботи програми, а клас AuthAspect дозволяє обмежувати доступ до певних методів та властивостей. Також, розробники можуть створити свої нащадки класу Aspect з потрібним функціоналом. Для реалізації не складних аспектів доречним буде FluentAspect, який дозволяє налаштувати наскрізну поведінку у Fluent стилі.

Для інтеграції на етапі виконання з використанням IOC, нащадок класу Introduce конфігурує контейнер створюючи потрібні перехоплювачі. Якщо конкретний IOC не підтримує перехоплювачі, то конфігурується проксі. Для інтеграції на етапі виконання без використання IOC виконується конфігурація фабрик. Клас CompileTimeIntroduction забезпечує модифікований компілятор необхідними даними для впровадження аспектів в потрібних точках коду в процесі компіляції.

Крім класів Aspect та Introduction необхідні сервісні класи, такі як AdviceConfigurationContext та PointcutConfigurationContex. Вони забезпечують гнучкий спосіб конфігурації поради та зрізу точок з'єднання відповідно. Для ініціалізації аспектів в програмі існує клас Bootstraper, який використовує класи ієрархії AspectProvider для отримання визначення аспектів, що зареєстровані в програмі. Кожен провайдер успадковується від AspectProvider та реалізує його метод GetAspect(). Наприклад, ConfigurationAspectProvider повертає аспекти, що зареєстровані за допомогою класу конфігурації, а AttributeAspectProvider повертає аспекти зареєстровані за допомогою атрибутів реєстрації аспектів.

Така архітектура забезпечує можливість створення нащадків класу Aspect та пов'язаних з ним класів конфігурації незалежно від нащадків класу Introduction. А це означає, що інтерфейс оголошення аспектів буде незалежним від способу впровадження.

## 5. Процес розробки системи

Є три основні архітектурні аспекти, які повинні бути визначені для забезпечення функціональних вимог, до початку розробки. Ці рішення:

1. Кордони (Boundaries).
2. Структура (Structure).
3. Модель домену (Domain model).

### 5. 1. Визначення кордонів

Перед створенням будь-яких артефактів (код або діаграми), архітектор повинен зрозуміти мету рішення і де це рішення підходить в контексті бізнес-проблеми. Це є вирішальним. Зокрема, архітектор повинен мати уявлення про те, які об'єкти виступають в якості входів до рішення, і які об'єкти потрібні для вирішення, щоб виконати бізнес-вимоги. Архітектор також повинен зрозуміти потік бізнес-операцій, тобто як вони виконуються вручну.

Знання кордонів рішення дає архітекторові розуміння цілі, до якої він може вести розробку. Це також забезпечує архітектору можливість взаємодіяти з замовником та перевірити правильність розуміння потреб. Це критичний перший крок до визначення структури рішення.

У системі впровадження аспектів в програмний код можна виділити такі вимоги:

– зміна поведінки коду без зміни самого коду;

– інкапсуляція наскрізного функціоналу у окремі сутності – аспекти;

– синтаксис оголошення аспектів не повинен залежати від методів впровадження;

– створення власних способів впровадження;

– створення власних варіацій синтаксису оголошення.

З цих вимог можна зробити висновок, що система має взаємодіяти з зовнішніми способами впровадження, які можуть являти собою компілятор, якщо впровадження відбувається на етапі компіляції або з механізмом прив'язки типів, у випадку впровадження на етапі виконання.

### 5. 2. Структурування рішення

Структурування рішення – одне з найважливіших завдань, що повинен вирішити архітектор. Не визначення структури рішення може призвести до його громісткості і викликати складнощі в розробці і підтримці. Рішення може забезпечувати виконання вимог, що не потрібні на поточний момент, але це дасть можливість розширення в майбутньому.

Одним з перших артефактів при виконанні структурування є схема концептуальної архітектури. Концептуальна архітектура визначає, як функціональні вимоги будуть виконані в структурі, визначеній для рішення. Визначення концептуальної архітектури повинно включати:

– як рішення піддається зовнішнім процесам, як користувачі будуть взаємодіяти з рішенням;

– як угоди будуть здійснюватися протягом всього рішення, як області рішення будуть доступні і піддаватися маніпуляціям;

– як рішення буде обробляти події всередині і зовні.

Визначаючи структуру рішення на початку процесу, архітектор може почати спілкуватися з різними зацікавленими сторонами. Бізнес спонсори часто використовують концептуальну схему, щоб почати створення маркетингу для рішення.

Використовуючи інструмент моделювання, який дозволяє архітекторові визначити концептуальну архітектуру при створенні робочий потоку рішення, можна звести до мінімуму будь-яку двозначність архітектури, і можна показати будь-які технічні питання, які можуть виникнути.

Після перевірки правильності кордонів, можна приступити до структуризації рішення. Від початкового аналізу, ми знаємо наступні вимоги рішення:

– існує необхідність інтеграції як на етапі компіляції, так і на етапі виконання;

– необхідно забезпечити незалежність оголошення від способу впровадження аспектів;

– способи впровадження можуть як поставлятись разом з фреймворком, так і у вигляді окремих бібліотек;

– користувач може розробити власні способи впровадження;

– синтаксис оголошення може бути різним, а саме з використанням наслідування та перевизначення методів аспектів, створення GENERIC аспектів та гнучка конфігурація (Fluent Api).

З цих потреб можна зробити висновок про необхідність розбиття системи на 2 модулі – оголошення та інтеграцію. Крім того, у випадку інтеграції на етапі компіляції необхідний окремий модуль, що є модифікацією компілятора. А у випадку інтеграції на етапі виконання необхідний модуль, що забезпечить створення перехоплювача чи проксі. З ядра також можна виділити модуль конфігурації, який може модифікуватися, для зміни синтаксису оголошення (рис. 2).

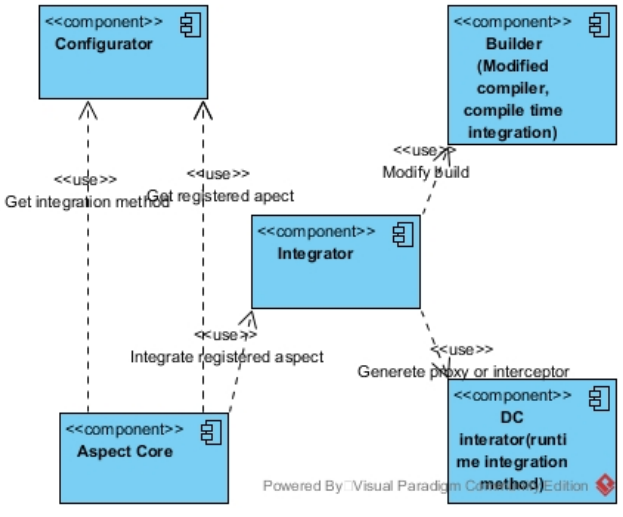


Рис. 2. Структурна схема системи

### 5. 3. Проектування моделі предметної області

Концептуальна архітектура є важливим першим артефактом в архітектурі рішення, але це тільки перший крок. Взявши концептуальну структурну схему за основу, її можна розширювати та деталізувати.

Модель предметної області містить об'єкти, які необхідні для виконання функціональних вимог. Модель також визначає відносини між об'єктами. Ця частина визначення вирішення також потребує архітектора розглянути, як контролювати доступ до об'єктів домену і як координувати операції в домені. Результат моделювання домену може включати в себе будь-яку або всі з наступних рівнів застосунку: frameworks, object models, service definitions.

Проаналізувавши функціональне навантаження на кожен з модулів, можна побудувати його структуру. Вона повинна забезпечувати зручне розширення системи та бути легкою в розумінні. На рис. 3 представлена діаграма класів системі інтеграції аспектів.

Опис класів:

1) Bootstrapper – точка входу в систему. Аспекти починають діяти після виклику методу Bootsprap.

2) IConfigurator – інтерфейс, що забезпечує отримання конфігурації. Configurator – його базова абстрактна реалізація.

3) Configuration – зберігає налаштування інтеграції та аспектів.

4) IaspectProvider – інтерфейс що забезпечує отримання списку аспектів. Його реалізації ConfigurationAspectProvider, ExternalConfigurationAspectProvider, AttributeAspectProvider забезпечують різні способи пошуку оголошення аспектів, відповідно: напряму в контексті конфігурації — AspectConfigurationContext, в зовнішньому джерелі (напр. XML файл конфігурації) та за допомогою атрибутів.

5) IAspect – інтерфейс, що представляє собою абстракцію сутності аспекту, та має два методи, які забезпечують отримання зрізу та поради. В системі визначено його 3 базові реалізації.

6) Aspect – абстрактна реалізація інтерфейсу Iaspect, наслідуючись від якого та визначаючи методи Advice і Pointcut можна оголосити свій аспект. Також позначивши нащадка атрибутом AutoRegAspect, можна забезпечити його автоматичне використання.

7) GenericAspect – узагальнений клас, що приймає аргументи типи, обмежені нащадками інтерфейсів IadviceProvider IpointcutProvider. Застосування цього класу як бази для аспектів доцільно тоді, коли існує необхідність розмежувати зріз та пораду.

8) FluentAspect – забезпечує створення аспекту під час виконання у гнучкому стилі. Може бути швидко застосований для невеликих аспектів, або для тимчасових аспектів в процесі налагоджування програми.

9) Advice – представляє собою пораду. Його методи виконуються у відповідних місцях кожної точки з'єднання.

10) ExecutionContext – представляє контекст виконання програми, з необхідною інформацією для методу поради.

11) Pointcut – представляє зріз, включає в себе набір точок з'єднання.

12) Joinpoint – точка з'єднання, описує, в якому місці програми необхідно впровадити аспект.

13) Introduction – клас, що забезпечує впровадження. Має один абстрактний метод Introduce, який при визначені в нащадку виконує впровадження.

14) CompileTimeIntroduction – є основою для методів впровадження на етапі компіляції.

15) ExecutionTimeIntroduction – є основою для методів впровадження на етапі виконання.

## 6. Інфрастуктура системи

В даній програмній системі база даних використовується для збереження налаштувань аспектів. Була створена наступна множина статичних класів (рис. 4).

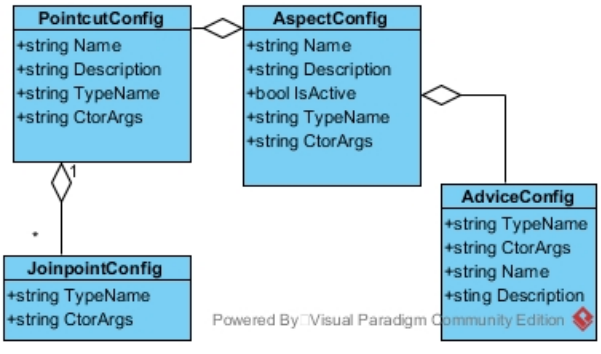


Рис. 4. Діаграма статичних класів

AspectConfig – клас для збереження конфігурації аспектів. Може зберігати або назву типу для створення відомого зрізу або PontcutConfig і AdviceConfig для конструювання динамічного аспекту.

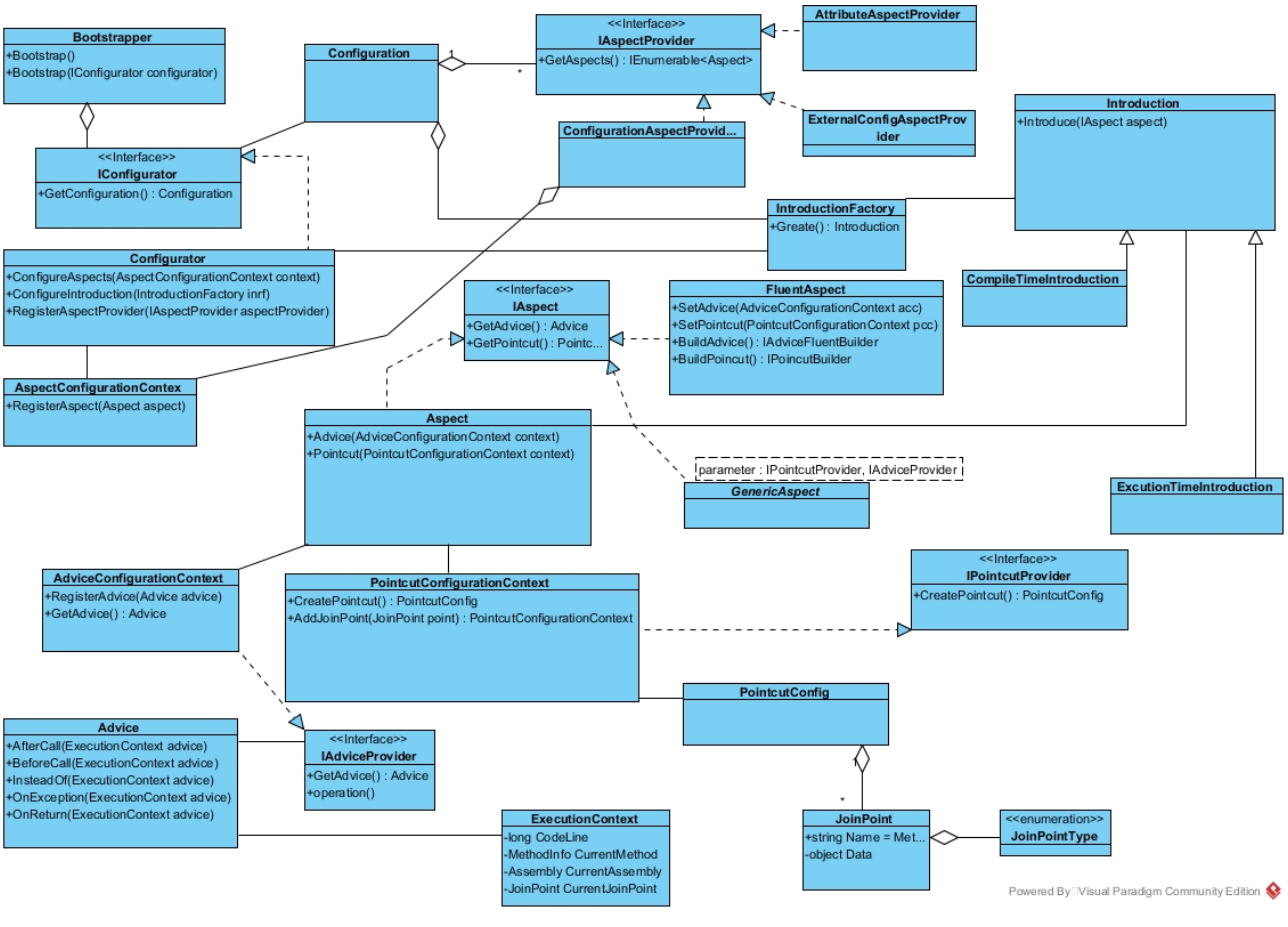


Рис. 3. Діаграма класів

PontcutConfig – клас призначений для збереження конфігурації зрізів. Може зберігати або назву типу для створення відомого зрізу або набір Joinpoint для конструювання динамічного зрізу.

JoinpointConfig – клас, призначений для збереження назви типу зрізу та параметрів, необхідних для створення об'єкту.

AdviceConfig – клас, призначений для зберігання назви типу поради та параметрів необхідних для створення об'єкту.

Для зберігання статичної інформації системи використовується реляційна база даних, так як вона забезпечує високий рівень незалежності структури збереження даних від структури програми. В процесі її розробки було побудовано концептуальну схему бази даних (рис. 5), а на її основі побудовано даталогічну модель (рис. 6).

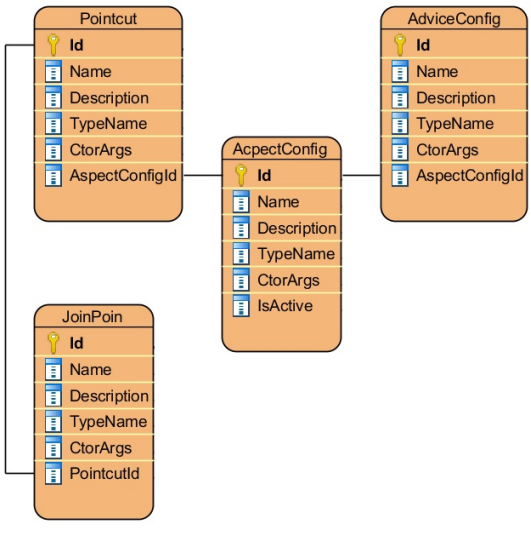


Рис. 5. Концептуальна схема бази даних


Рис. 6. Даталогічна схема бази даних

У табл. 1–4 представлено опис полів бази даних.

База даних використовує лише стандартні SQL типи даних та не має збережених процедур чи функцій, що дозволяє розміщувати її на будь-якому сучасному SQL сервері. Крім того, існує можливість перевизначити провайдер даних та використати для збереження інформації NoSQL базу чи інші типи постійних сховищ.

Таблиця 1

Структура таблиці AspectConfig

| Назва стовпчика | Тип даних | Опис |
|---|---|---|
| Id | integer(10) | Унікальний ідентифікатор |
| Name | Varchar(255) | Логічна назва аспекта (напр. аспект ведення протоколу роботи) |
| Description | Varchar(MAX) | Опис аспекту |
| TypeName | Varchar(255) | Повна назва системного типу аспекту для рефлексійного створення |
| CtorArgs | Varchar(MAX) | Агрументи конструктора для ініціалізації типу аспекта |
| IsActive | Bit | Прапорець. Показує чи активний данний аспект |

Таблиця 2

Структура таблиці PointcutConfig

| Назва стовпчика | Тип даних | Опис |
|---|---|---|
| Id | integer(10) | Унікальний ідентифікатор |
| Name | Varchar(255) | Логічна назва зрізу (напр. методи, назва яких закінчується на AppService та методи що позначені атрибутом Log) |
| Description | Varchar(MAX) | Опис зрізу |
| TypeName | Varchar(255) | Повна назва системного типу зрізу для рефлексійного створення |
| CtorArgs | Varchar(MAX) | Агрументи конструктора для ініціалізації типу зрізу |

Таблиця 3

Структура таблиці JoinPointConfig

| Назва стовпчика | Тип даних | Опис |
|---|---|---|
| Id | integer(10) | Унікальний ідентифікатор |
| Name | Varchar(255) | Логічна назва точки впровадження (напр. поля з навою User) |
| Description | Varchar(MAX) | Опис точки впровадження |
| TypeName | Varchar(255) | Повна назва системного типу точки впровадження для рефлексійного створення |
| CtorArgs | Varchar(MAX) | Агрументи конструктора для ініціалізації типу точки впровадження |

Таблиця 4

Структура таблиці AdviceConfig

| Назва стовпчика | Тип даних | Опис |
|---|---|---|
| Id | integer(10) | Унікальний ідентифікатор |
| Name | Varchar(255) | Логічна назва поради (напр. записати назву методу в файл) |
| Description | Varchar(MAX) | Опис поради |
| TypeName | Varchar(255) | Повна назва системного типу поради для рефлексійного створення |
| CtorArgs | Varchar(MAX) | Агрументи конструктора для ініціалізації типу поради |

## 7. Модулі та розгортання системи

### 7. 1. Діаграма компонентів

Головним компонентом системи є бібліотека Saspect.dll, яка використовуючи компонент Bootstrapper впроваджує в програму клієнта аспекти. Для налаштування аспектів використовується Configurator, що забезпечує отримання конфігурації з вказаним провайдером аспектів та метод інтеграції. Метод інтеграції реалізований у вигляді окремого компоненту та завантажується динамічно, саме такий, який було сконфігуровано. Для деяких методів інтеграції необхідний компонент компілятора (SaspectBuilder) (рис. 7).

### 7. 2. Діаграма розташування

Зазвичай вся система розташовується на комп'ютері розробника-користувача фреймворку. Якщо при розробці програм застосовується сервер побудови програм, то система має бути розміщена також і на сервері (рис. 8). У випадку використання інтеграції на етапі компіляції, модифікований компілятор також має бути розміщений на комп'ютері розробника-користувача та на сервері.

З діаграм розміщення можна зробити висновок, що розгортання системи є відносно простим, особливо у випадку використання інтеграції на етапі виконання.

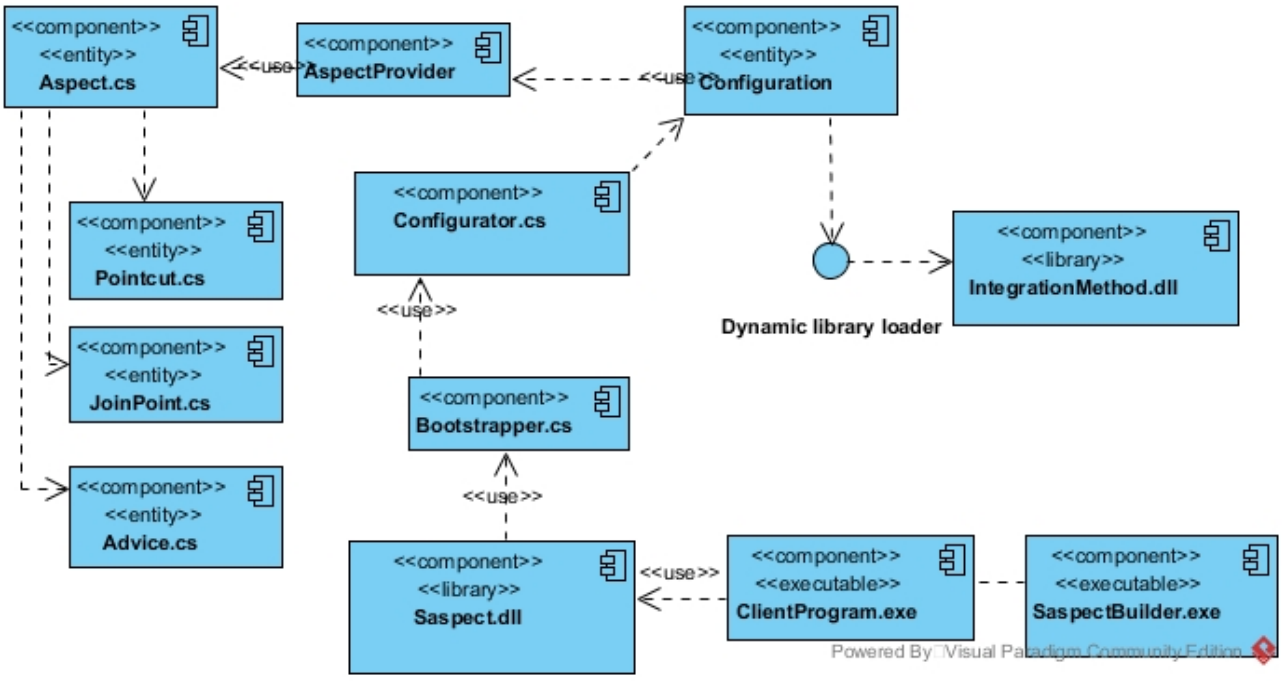


Рис. 7. Діаграма компонентів фреймворку інтеграції аспектів

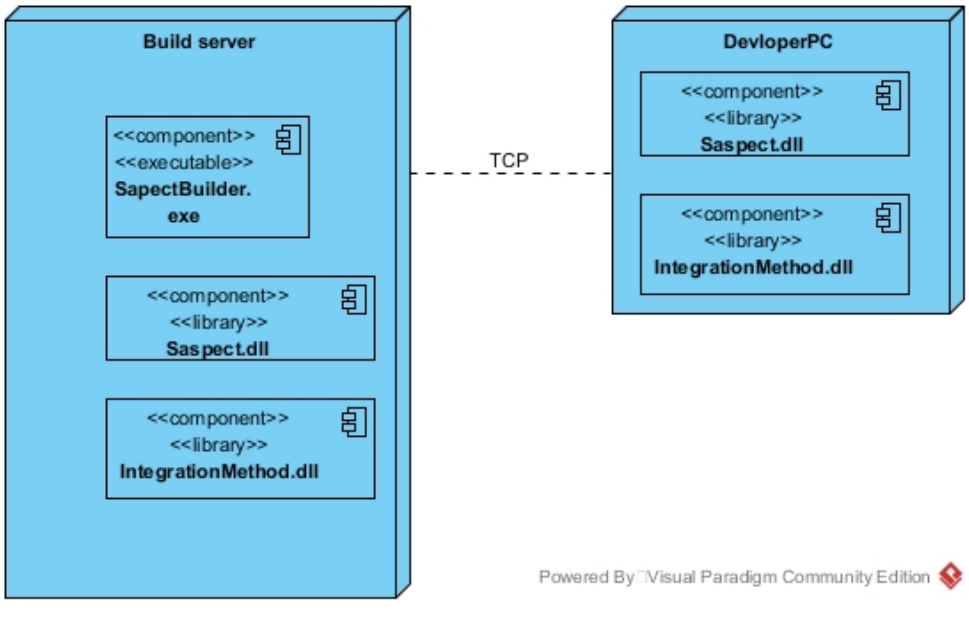


Рис. 8. Діаграма розташування фреймворку інтеграції аспектів

### 8. Приклад використання фреймворку

Нехай маємо типову задачу: необхідно створити WEB-сервіс, що забезпечує отримання, створення, оновлення та видалення користувачів певної системи. З використанням сучасних технологій побудови веб API та ORM фреймфорка, на перший погляд код сервісу буде простим. Розроблено демонстраційний сервіс на мові програмування C# та з використанням технології WebApi. (рис. 9).

Але у реальних задача виникає потреба в реалізації такого функціоналу, як ведення журналу, обробки помилок, обмеження доступу, перевірки на коректність для аргументів та інші типові задачі [12]. В результаті, наприклад, метод Get, що займав один рядок коду, значно збільшився в обсязі і це ускладнило його розуміння (рис. 10). Крім того, у разі необхідності змінити поведінку, наприклад ведення журналу, може виникнути необхідність змінювати виклики в усіх місцях, де він застосовується.

Використання аспектів дозволяє виділити наскрізний функціонал в окремі сутності. Так, з використанням аспектів, метод Get залишається таким же компактним, як на рис. 9, і, в той же час, має функціонал, як на рис. 10. Саме ж оголошення поведінки аспектів зібране о одній сутності, що дозволяє легко розуміти та модернізовувати код програми (рис. 11).

```
public class UserService
{

        private Context context = new Context();

        public User Get(int id)
        {
                return context.Users.FirstOrDefault(r => r.id = id);
        }

        public User Create(User user)
        {
                return context.User.Insert(user);
        }

        public User Update(User user)
        {
                context.User.Attach(user);
                contex.Entry(user).State = EntityState.Modified;
                return user;
        }

}
```

Рис. 9. Базова реалізація сервісу обслуговування користувача

```
public User Get(int id)
{
        if(id < 0)
        {
                Log.Warn("Bad input value - id");
                throw new ArgumentException("Id must be > 0");
        }
        if(!User.Current.HasPermission("GetUser"))
        {
                Log.Warn("User {0} have not permission.",User.Current);
                throw new PermissionMissingException("You can not get user");
        }
        try
        {
                Log.Trace("Begin execution getting user with id = {0}", id);
                var result = context.User.FirstOrDefault(r => r.id == id);
                Log.Trace("Execution getting user with id = {0} and return {1}",id,result);
                return result;
        }
        catch(SQLException ex)
        {
                Log.Error("SQL exception on user get", ex);
                HandleSQlException(ex);
        }
        catch(Eception ex)
        {
                Log.Error("Unknown exception in user get", ex);
                HandleException(ex);
        }
        finally
        {
                context.Dispose();
        }
}
```

Рис. 10. Код методу Get з додатковим функціоналом

```
public class ExceptionHandlerAspect: Aspect
{
        public ovverider void Pointcut(PointcutConfigurationContext context)
        {
                contxt.CreateJoinPoit().OnMethodCall().InClass(r => r.Name.EndWith("Service"));
        }
        public override void Advice(AdviceConfigurationContext context)
        {
                context.OnException(ec =>
                {
                        if(ec.Exception is SqlException)
                        {
                                Log.Error("SQL Exception on {1}",ec.MethodName);
                        }
                        else
                        {
                                Log.Error("Unknown exception on {1} {2}",ex.MethodName,ec.Exception);
                        }
                }

}}
```

Рис. 11. Аспект обробки виключень у методах веб сервісу

Одного разу оголошений аспект відпрацьовує в усіх налаштованих місцях, що значно зменшує обсяг коду (табл. 5), особливо на складних програмних системах, які є характерними в Enterprise сегменті [13]. Також використання аспектів є надзвичайно ефективним при вирішені задач моделювання, де в наявності складна і мінлива логіка та структура програми та стабільний сервісний код.

Таблиця 5

Кількість рядків сирцевого коду

| Кількість рядків сирцевого коду до впровадження аспектів | Кількість рядків сирцевого коду після впровадження аспектів |
|---|---|
| 5000 | 4228 |
| 10000 | 7906 |
| 15000 | 11584 |
| 20000 | 15262 |
| 25000 | 18940 |
| 30000 | 22618 |
| 35000 | 26296 |
| 40000 | 29974 |
| 45000 | 33652 |
| 50000 | 37330 |

Можна зробити висновок, що використання аспектів дозволяє досягти значного скорочення сирцевого коду, особливо для великих за обсягом програм.

## 9. Висновки

1. Розроблено архітектурне рішення, яке забезпечує можливість оголошення аспектів в незалежності від способу їх інтеграції. Для досягнення незалежності використано шаблон проектування міст та ряд породжуючих шаблонів, таких як фабричний метод, будівельник та абстрактна фабрика.

2. Розроблено три способи оголошення аспектів, а саме: оголошення за допомогою наслідування від базового класу, узагальнення шаблонного класу, та гнучке створення аспекту на етапі виконання.

3. Для інтеграції на етапі компіляції був розроблений спеціальний модуль інтеграції та модифікація компілятора Roslyn, яка забезпечує виконання системи конфігурації аспектів та впроваджує в код точки виклику порад.

4. Для інтеграції на етапі виконання без використання контейнера залежностей розроблено методи-помічники для створення проксі класів. Також розроблено модулі для популярних контейнерів залежностей, які дозволяють виконати інтеграцію засобами цих контейнерів.

5. Проведено тестування розробленої системи, яке показало значне скорочення розміру сирцевого коду. Найбільш виражене скорочення було у великих системах рівня підприємства. При використанні впровадження на етапі компіляції падіння швидкодії програм не виявлено. При використанні інтеграції на етапі виконання втрати швидкодії не перевищують таких при використанні аналогічного проксі класу.

Література


1. Floyd, R. The Paradigms of Programming [Text] / R. W. Floyd // Communications of the ACM. – 1979. – Vol. 22, Issue 8. – P. 455–460. doi: 10.1145/359138.359140
2. Бадд, Т. Объектно-ориентированное программирование в действии. [Текст] / Т. Бадд. – СПб.: «Питер», 1997. – 464 с.
3. Гамма, Е. Design Patterns: Elements of Reusable Object-Oriented Software [Text] / Е. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. – СПб.: Питер, 2014. – 372 с.
4. Miles, R. AspectJ Cookbook [Text] / R. Miles. – O'Reilly Media, 2012. – 356 p.
5. Нейгел, К. C# 5.0 и платформа .NET 4.5 для профессионалов [Текст] / К. Нейгел, Б. Ивьен, Д. Глинн, К. Уотсон, М. Скиннер. – М. : ООО «И.Д. Вильямс», 2013. – 1543 с.
6. Gael, F. PostSharp Roadmap and Support Policies Published [Electronic resource] / F. Gael. – PostSharp Blog, 2015. – Available at: http://www.postsharp.net/blog/post/PostSharp-Roadmap-and-Support-Policies-Published

7. Yang, H. Software Reuse in the Emerging Cloud Computing Era [Text] / H. Yang, X. Liu. – Information Science Reference, 2012. – 54 p. doi: 10.4018/978-1-4666-0897-9
8. Sells, C. Essential.NET: The common language runtime [Text] / C. Sells. – Addison-Wesley Professional, 2011.
9. Gael, F. Dino Esposito, Cutting Edge – Aspect-Oriented Programming, Interception and Unity 2.0 [Electronic resource] / F. Gael. – MSDN Magazine, 2013. – Available at: https://msdn.microsoft.com/en-us/magazine/gg490353.aspx
10. Rossi, J. Introduction to AOP With Castle [Electronic resource] / J. Rossi. – Castle Project Blog, 2015. – Available at: http://docs.castleproject.org/Default.aspx?Page=Introduction-to-AOP-With-Castle&NS=Windsor&AspxAutoDetectCookieSupport=1
11. Win, B. Security through aspect-oriented programming [Text] / B. Win, B. Vanhaute, B. De Decker. – In Advances in Network and Distributed Systems Security, 2002. – P. 125–138. doi: 10.1007/0-306-46958-8_9
12. Kiczales, G. Aspect-oriented programming [Text] / G. Kiczales, J. Lamping, A.Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, J. Irwin // ECOOP'97. Proceedings of the 11th European Conference on Object-Oriented Programming, 1997. – P. 220–242. doi: 10.1007/BFb0053381
13. Fowler, M. Patterns of Enterprise Application Architectur [Text] / M. Fowler. – Addison Wesley, 2012.



***Показано, що причиною високого відсотку бракованих гумометалевих виробів на виході технологічного процесу їхнього виготовлення є ігнорування на етапі проектування зв'язності параметрів підсистем окремо всередині конструкції і технології, а також між цими підсистемами. Побудована комплексна оптимізаційна модель конструкції та технології. Запропонована комплексна символьна модель для еволюційної оптимізації за допомогою генетичного алгоритму***



***Ключові слова: гумометалеві вироби, зв'язність параметрів, генетичні алгоритми, комплексні символьні моделі***



***Показано, что причиной большого процента бракованных резинометаллических изделий на выходе технологического процесса их изготовления является игнорирование на этапе проектирования связности параметров подсистем отдельно внутри конструкции и технологии, а также между этими подсистемами. Построена комплексная оптимизационная модель конструкции и технологии. Предложена комплексная символьная модель для эволюционной оптимизации с помощью генетического алгоритма***



***Ключевые слова: резинометаллические изделия, связность параметров, генетические алгоритмы, комплексные символьные модели***

УДК 004.942:62-272.6
DOI: 10.15587/1729-4061.2016.65456


# ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ОПТИМІЗАЦІЇ КОНСТРУКЦІЇ ТА ТЕХНОЛОГІЇ ВИГОТОВЛЕННЯ ГУМОМЕТАЛЕВИХ ВИРОБІВ


**О. С. Савєльєва**
Доктор технічних наук, доцент*
E-mail: okssave@gmail.com
**І. І. Становська**
Кандидат технічних наук**
E-mail: iraidasweet07@rambler.ru
**О. Ю. Лебедєва***
E-mail: ozrti@rambler.ru
**А. В. Торопенко**
Кандидат технічних наук*
E-mail: alla.androsyk@gmail.com
*Кафедра нафтогазового та хімічного машинобудування***
**Кафедра вищої математики та моделювання систем***
***Одеський національний політехнічний університет
пр. Шевченка, 1, м. Одеса, Україна, 65044


## 1. Вступ

При проектуванні конструкції і технології виготовлення окремо гумових або окремо металевих виробів, як правило, застосовуються різні методи оптимізації, що обумовлюється суттєво різними властивостями цих матеріалів та очевидними відмінностями в умовах їхнього виготовлення та експлуатації, а також в зовнішніх прихованих впливах на виріб. Натомість, об'єднані гумометалеві вироби (механічні амортизатори, автомобільні покришки, корпуси підводних човнів, магнітна та електропровідна гуми, тощо) [1–4], хоча й складаються із суттєво різнорідних за властивостями елементів (металу та гуми), проектуються, виготовляються і працюють «як одна деталь», і отже вимагають при цьому деякого сумісного, комплексного підходу на усіх перелічених етапах свого життєвого циклу.