УДК 681.03

В.М. Грищенко

АЛГЕБРАЇЧНА МОДЕЛЬ РЕФАКТОРІНГУ КОМПОНЕНТІВ

Наведено алгебраїчний підхід до формалізації рефакторінгу компонентів. На базі моделі компонента разглянуті методи рефакторінгу, припустимі операції та умови цілісності компонентів. Визначені концепції побудови алгебраїчної моделі рефакторінгу компонентів та описана внутрішня компонентна алгебра.

Вступ

Формальні, у тому числі й алгебраїчні, методи широко застосовуються в сучасних дослідженнях із проблем компонентного програмування. Як відзначено в [1], компонентний підхід — це нова реалізація концептуальної ідеї композиційного програмування на сучасному етапі і тому значний обсяг теоретичних результатів із проблем композиції став основою для побудови теорії компонентного програмування.

Серед таких результатів можна виділити наступні класи моделей, методів, підходів, що призначені для аналізу різних аспектів компонентної теорії [1, 2]:

— мови і моделі опису інтерфейсів;

— мови опису взаємодії модулів та об'єктів;

— шаблони проектування;

— мови опису компонентних архітектур;

— компонентні моделі.

Їх особливість полягає у тому, що вони орієнтовані на розгляд компонента як цільного, формально поданого об'єкта, що має визначені характеристики і типову структуру. Це відображає ідеалізовану точку зору у компонентному програмуванні, суть якої полягає в тому, що компоненти є базовими елементами (блоками, модулями і т.д.), на основі яких будується компонентна програма. При цьому такі елементи мають атомарну структуру і цілком визначені і наперед задані властивості та характеристики.

Практика компонентного програмування показала, що така точка зору дуже обмежена й у багатьох випадках базові компоненти піддаються деякій попередній обробці — зміні інтерфейсів і деяких аспектів реалізації, зміні параметрів конфігурації і розгортання (процедури інсталяції та прив'язки компонента до середовища) і т.д. При цьому отримані об'єкти розглядаються у якості нових компонентів, які входять до складу компонентних програм. Проте операції над базовими компонентами носять не довільний характер, а задовольняють визначеним правилам та умовам. Суть їх полягає у тому, що отриманий у результаті виконання певної операції об'єкт теж повинен бути компонентом із усіма необхідними атрибутами і властивостями. Це викликано існуванням умов, що накладаються компонентними моделями та правилами побудови компонентних структур і програм у сучасних середовищах.

Таким чином, поступово формується окрема галузь досліджень у компонентному програмуванні — рефакторінг компонентів, що включає сукупність моделей, методів і засобів перетворення, а також зміни структурних і якісних характеристик базових об'єктів для одержання нових компонентів, найбільш придатних для побудови цільової компонентної програми.

У цій статті мова йде про формальне подання методів рефакторінгу компонентів, кінцева мета якого складається у побудові відповідної алгебраїчної моделі або внутрішньої алгебри компонентів (зовнішні компонентні алгебри, як було відзначено вище, розглядають компоненти як цільні об'єкти компонентних конфігурацій і структур [3]).

Визначення базових понять і термінів

Базова термінологія для компонентного програмування і компонентних програм, що концептуально зв'яза-

на з даною роботою, наведена у [3]. Основні терміни для визначення компонента, його властивостей, характеристик, структури приведені у багатьох публікаціях, наприклад у [4, 5]. Введемо основні поняття і терміни, безпосередньо пов'язані з рефакторінгом компонентів. До них, зокрема, відносяться поняття компонента і саме рефакторінгу.

Програмний компонент чи просто **компонент** — це незалежний від мови програмування, самостійно реалізований програмний об'єкт, що забезпечує виконання певної множини програмних сервісів і поданий як взаємозамінний контейнер, доступ до якого можливий тільки за допомогою інтерфейсів, що визначають його функціональні можливості і порядок звертання до його операцій.

Як вже відзначалося вище, у компонентному програмуванні компонент є неподільною і інкапсульованою сутністю, що задовольняє визначеним функціональним вимогам, а також вимогам архітектури, структури й організації взаємодії в компонентній програмі.

Рефакторінг — це сукупність моделей, методів, процесів, які застосовуються до визначених класів об'єктів з метою одержання нових або зміни існуючих структурних і функціональних показників об'єктів, що забезпечує їм нові якісні характеристики та поведінку.

Найбільший розвиток напрямок рефакторінгу здобув у об'єктно-орієнтованому програмуванні [6], зокрема, у зв'язку із застосуванням шаблонів проектування [7] та іншими методами поліпшення коду і структури об'єктно-орієнтованих програм. Розроблено цілі бібліотеки типових трансформацій об'єктів (класів), що поліпшують ті чи інші характеристики [6].

На противагу цьому рефакторінг компонентів — порівняно нова галузь дослідження і говорити про значні результати в ній поки що рано. Найбільш часто зустрічаються публікації з практичної тематики, наприклад практичні методики змін характеристик компонентів у рамках конкретних компонентних моделей. Почасти це викликано тим, що компоненти, як структурні утворення, значно складніше і багатогранніші, ніж об'єкти і класи в об'єктно-орієнтованому програмуванні. Тому рефакторінг компонентів як самостійний напрямок досліджень знаходиться ще в стадії становлення.

Рефакторінг компонента — це процес зміни існуючого компонента з метою надання йому нових функціональних і структурних характеристик, що найбільш повно задовольняють вимогам компонентної конфігурації, до складу якої шуканий компонент повинен увійти. Фактично це означає таку зміну існуючого компонента, за якої він найбільше повно відповідає результатам проектування компонентної програми.

Метод рефакторінгу компонента — це цільовий спосіб одержання нового компонента на базі існуючого, що визначає операції модифікації (зміни, заміщення, розширення) елементів, які входять до його складу. Кожен метод характеризується метою (яким вимогам повинен задовольняти отриманий компонент) і можливими послідовностями операцій перетворення складових компонента.

Операція рефакторінгу — базова, атомарна функція перетворення, що зберігає цілісність компонента.

Цілісність компонента — сукупність правил, обмежень і залежностей між його складовими елементами, що дозволяє розглядати компонент як єдину й цільну структуру з властивостями та характеристиками, які відповідають визначенню класу компонентів для компонентного програмування.

Наведені вище визначення є базовими для формального подання рефакторінгу компонентів. Визначення допоміжних термінів і понять буде вводитись у процесі подання матеріалу відповідно до контексту викладу.

Модель компонента

Під моделлю компонента розуміється абстрактне подання, що відобра-

жає його основні властивості, характеристики, типову структуру, а також здатність до взаємодії з іншими елементами компонентного середовища. За основу узятий опис та подання моделі з [3].

Модель довільного компонента у загальному випадку представляється наступною сукупністю:

$$\mathbf{Comp = (CName, CInt, CFact, CImp, CServ)}, \quad (1)$$

де **CName** — унікальне ім'я компонента;

CInt = {CInt^i} — множина інтерфейсів, пов'язаних з компонентом;

CFact — інтерфейс керування екземплярами компонента;

CImp = {CImp^j} — множина реалізацій компонента;

CServ = {CServ^r} — інтерфейс, що визначає множину системних сервісів, які необхідні для підтримки функціонування компонента і взаємодії з компонентним середовищем.

Ім'я компонента унікальне у будь-якому просторі імен для компонентного середовища. Для вирішення можливих колізій застосовується метод кваліфікованих імен. Фактично це означає, що кожне ім'я входить у деякий окремий простір імен, сукупність яких визначає простір імен компонентного середовища. Керування такими просторами забезпечується спеціальними алгоритмами побудови просторів імен у вигляді ієрархічного дерева, де кожна вершина однозначно визначається маршрутом (послідовністю вершин) від кореня дерева.

Множина **CInt = CIntI ∪ CIntO** складається з інтерфейсів двох типів. До першого типу (множина **CIntI**) відносяться інтерфейси, які реалізуються в середовищі цього компонента, тобто мають відповідні реалізації методів. До другого типу (множина **CIntO**) відносяться інтерфейси, реалізовані в інших компонентах, але функціональність яких потрібна для виконання методів цього компонента.

Кожен інтерфейс компонента поданий у такому вигляді:

$$\mathbf{CInt^i = (IntName^i, IntFunc^i, IntSpec^i)}, \quad (2)$$

де **IntName^i** — ім'я інтерфейсу;

IntFunc^i — функціональність, яку визначає цей інтерфейс (сукупність методів);

IntSpec^i — специфікація інтерфейса (описи типів даних, констант, інших елементів даних, сигнатур методів і т.д.).

Provide(CInt^i) позначається реалізація методів, визначених інтерфейсом **CInt^i**, яка надається деякою компонентною реалізацією (програмний елемент, що безпосередньо забезпечує виконання методів інтерфейсів). Функціональне позначення для цієї залежності обрано у зв'язку з тим, що в компонентному програмуванні інтерфейс компонента може визначатись окремо від його реалізації. Тобто на певних етапах розробки (проектування) компонентної програми для обраних інтерфейсів застосовуються існуючі компоненти, які надають свої функціональні сервіси для реалізації цього інтерфейса [8].

Інтерфейс **CFact** визначає методи, необхідні для керування екземплярами компонента. Зокрема, до них відносяться:

— пошук та визначення місця знаходження необхідного екземпляра компонента **Locate**;

— створення нового екземпляра компонента **Create**;

— завершення функціонування та видалення екземпляра компонента **Remove**.

Ці методи складають основу для будь-яких інтерфейсів керування екземплярами у рамках будь-яких компонентних моделей. Тому в найбільш загальному випадку маємо

$$\mathbf{CFact = \{Locate, Create, Remove\}}. \quad (3)$$

Кожна реалізація компонента описується наступним чином:

$$\mathbf{CImp^j = (ImpName^j, ImpFunc^j, ImpSpec^j)}, \quad (4)$$

де **ImpName^j** — ідентифікатор чи ім'я реалізації компонента;

ImpFunc^j — функціональність, задіяна даною реалізацією (сукупність реалізацій методів);

ImpSpec^j — специфікація реалізації (опис умов виконання, опис параметрів настроювання реалізації і т.д.).

Реалізація являє собою сукупність методів визначеної сигнатури і типів даних для вхідних та вихідних параметрів. Відповідно до цих сигнатур і типів даних відбувається процес зіставлення реалізацій і інтерфейсів, що містять опис методів, які входять до їх складу. Цей процес називається процесом зв'язування. На відміну від об'єктно-орієнтованого й інших підходів до програмування зв'язування в компонентному програмуванні виконується на більш пізніших стадіях, тобто на заключних етапах побудови компонентної програми, а іноді й під час виконання, як, наприклад, для динамічних інтерфейсів CORBA [9].

Множину системних сервісів **CServ = {CServ^r}** складають сервіси, необхідні для організації побудови та функціонування компонентних середовищ, а також керування компонентними конфігураціями. Зокрема, мінімально необхідний набір сервісів включає:

1. Сервіс найменування **Naming**. Забезпечує можливість пошуку компонентів у розподіленому середовищі відповідно до визначених просторів імен та їх місця знаходження.

2. Сервіс зв'язування **Binding**. Необхідний для визначення (зв'язування) відповідності "ім'я — об'єкт" і застосовується до компонентів, знайдених у компонентному середовищі.

3. Сервіс транзакцій **Transaction**. Забезпечує організацію і керування функціонуванням сукупності компонентів, що розглядається як окрема транзакція.

4. Сервіс повідомлень **Messaging**. Необхідний для організації спілкування між компонентами і є складовим елементом у моделі асинхронних транзакцій.

У реальних компонентних середовищах можуть бути реалізовані й інші системні сервіси, наприклад керування подіями, служба каталогів тощо. Але часто такі сервіси визначаються умовами спрощення організації компонентних середовищ і можуть бути утворені на базі інших сервісів. Надалі будемо вважати, що перераховані вище чотири сервіси є обов'язковими для будь-якої компонентної моделі і її реалізації. Вони відображають реалізацію базових функцій керування компонентними середовищами:

— пошук компонентів;

— доступ до їх ресурсів;

— організація обміну інформацією між компонентами;

— керування функціонуванням динамічно визначеної сукупності компонентів.

Необхідною вимогою щодо припустимого подання компонента є умова цілісності:

$$(\forall CInt^i \in CIntI)\,(\exists CImp^j \in CImp)\ \mathbf{Provide}(CInt^i) \subseteq CImp^j. \qquad (5)$$

Слід уточнити наявність знака включення в даній формулі. Це означає, що обрана реалізація може забезпечити підтримку не тільки необхідного інтерфейсу, але й інших. Практичні технології і мови програмування (CORBA, Java, C++ і ін.) дозволяють створювати такі реалізації та містять для цього необхідні засоби. Також відзначимо, що для кожного з таких інтерфейсів може існувати кілька реалізацій, які відрізняються особливостями функціонування (наприклад, операційним середовищем, засобами збереження даних тощо).

Загальна класифікація методів рефакторінгу у компонентному програмуванні

Для проведення класифікації певної множині об'єктів можуть бути обрані різні класифікаційні ознаки, які відображають різні аспекти і точки зору на множину, що піддається класифікації. Для компонентного програмування існують дві наперед визначені класифікації методів рефакторінгу.

1. Відповідно до життєвого циклу у компонентному програмуванні (рефакторінг у більш широкому розумінні). У [3] наводиться загальна характеристика життєвого циклу для компонентного програмування з описом окремих етапів і об'єктів компонентного середовища. Рефакторінг у такому контексті означає, що для застосування деякого компонента більш доцільним може бути шлях зміни елементів середовища, ніж модифікація самого компонента. Наприклад, більш простим методом може бути включення нового сервера для обраного компонента з наступною інтеграцією до складу середовища, ніж його переробка під існуючі сервери. Ця класифікація визначає методи, які пов'язані:

— зі зміною серверної складової компонентного середовища (додавання нового сервера, заміна існуючого, зміна параметрів конфігурації і т.д.);

— зі зміною контейнерної складової компонентного середовища (додавання нового контейнера, заміна існуючого, зміна параметрів конфігурації, розміщення в різних серверах з різними конфігураціями і т.д.);

— з рефакторінгом компонента (зміна інтерфейсів, реалізацій і т.д.);

— з процесом настроювання компонентів на параметри компонентного середовища (розгортання, зміна параметрів конфігурації);

— з процесом настроювання клієнтської частини для взаємодії з компонентами (заміна клієнтської частини, зміна протоколу взаємодії з компонентом і т.д.);

— з керуванням компонентним середовищем і компонентними конфігураціями (зміна складу чи конфігурації середовища залежно від сумісності та можливостей спільного функціонування окремих компонентів).

2. Класифікація, що безпосередньо пов'язана з рефакторінгом компонентів, базується на типовій структурі та моделі компонента, яка подана вище, і поділяється:

— на рефакторінг інтерфейсів (додавання, розширення і т.д.) або реалізацій (додавання, розширення і т.д.);

— зміну інтерфейса керування екземплярами компонентів (додавання нових функцій) або системними сервісами (додавання нових функцій).

Надалі під методами рефакторінгу будуть розумітися тільки методи з другої класифікації.

Припустимі операції рефакторінгу компонентів

Не всі операції над компонентами припустимі у процесі формалізації моделі рефакторінгу. Серед множини операцій необхідно виділити тільки ті, які задовольняють наступним умовам:

— об'єкт, отриманий у результаті рефакторінгу, повинен бути компонентом, тобто мати властивості, характеристики, а також типову структуру, що є основою визначення множини компонентів;

— у процесі рефакторінгу компонент не повинен утрачати свою функціональність, тобто зберігати можливість застосування в раніше побудованих компонентних конфігураціях і програмах;

— компонент, отриманий у результаті рефакторінгу, повинен відповідати вимогам і обмеженням компонентної моделі, у якій застосовувався базовий компонент.

З приведених умов випливають наступні висновки:

— операції видалення складових елементів компонента неприпустимі;

— операції зміни складових елементів компонента припустимі за умови збереження існуючої функціональності і порядку звертання до неї;

— компонент, отриманий у результаті рефакторінга, повинен застосовуватись у тій ж компонентній моделі, що й базовий.

Існує досить складна і важлива проблема, пов'язана з іменуванням і ідентифікацією нових компонентів. Методи рефакторінгу можуть бути різними, і, відповідно до визначення, нові компоненти повинні однозначно ідентифікуватися. З огляду на те, що таких

компонентів може бути декілька, необхідно визначити правила іменування нових компонентів і дисципліну їхнього застосування в компонентних конфігураціях (як нових, так і існуючих). Розв'язок цієї проблеми торкається задачі керування компонентними конфігураціями, яка вирішується на основі використання зовнішньої компонентної алгебри [3]. З метою спрощення викладу у цій статті для визначеності приймається обмеження (яке не впливає на сутність викладених результатів), що в результаті застосування операцій рефакторінгу отримується компонент з іншим ім'ям.

На основі наведеного вище визначаються наступні припустимі операції рефакторінгу компонентів:

— додавання нової реалізації для існуючого інтерфейсу;

— додавання нової реалізації для нового інтерфейсу;

— розширення існуючої реалізації;

— заміна існуючої реалізації новою з еквівалентною функціональністю;

— додавання нового інтерфейсу (за умови наявності відповідної реалізації);

— розширення існуючого інтерфейсу (за умови наявності відповідної реалізації);

— розширення інтерфейсу **CFact** (за умови існуванні реалізації для нових методів керування екземплярами компонентів);

— розширення інтерфейсу **CServ** (за умови існуванні реалізації для нових системних сервісів у компонентному середовищі).

Основні концепції внутрішньої компонентної алгебри

Концептуальну основу внутрішньої компонентної алгебри складають:

1. Модель компонента, що визначається виразом (1) і є формальною основою алгебри. Сутність даної концепції зводиться до наступних аксіом:

— операції алгебри визначаються на елементах цієї моделі і тільки на них;

— у результаті застосування операцій алгебри завжди виконуються необхідні умови цілісності компонента (5).

Фактично це означає, що складові моделі (1) визначають основу множини елементів для алгебри, а умова (5) — частковий характер окремих операцій над цією множиною.

2. Операції внутрішньої компонентної алгебри є формалізацією методів та операцій рефакторінгу компонентів на певному рівні абстракції без урахування аспектів реалізацій цих методів і їхньої специфіки в алгебрі. Зокрема, це стосується умови цілісності компонентів. При побудові алгебри важливою є умова самої цілісності, а не методи її встановлення.

3. Розглядаються тільки ті операції, котрі пов'язані з припустимими методами рефакторінгу. При цьому обов'язковою умовою є забезпечення цілісності компонента.

4. Структура і визначення внутрішньої компонентної алгебри узгоджена з іншими формальними моделями компонентного програмування. Суть концепції складається у побудові єдиного формального апарата компонентної теорії. Зокрема, у [3] описується зовнішня компонентна алгебра, що складає основу формального апарата для компонентних конфігурацій, середовищ і програм. Внутрішня компонентна алгебра узгоджена з нею на концептуальному та модельному рівнях.

Структура внутрішньої компонентної алгебри

Розглянемо важливе зауваження щодо внутрішньої структури компонента. Сучасні мови і моделі для практичного застосування компонентного підходу (наприклад, Java [10, 11], CORBA [11, 12]), як правило, вимагають, щоб кожна реалізація явно вказувала інтерфейс, для якого вона створена. Необхідність такої схеми обумовлена, в основному, практичними розрахунками — перевірка сигнатур методів і відповідності типів даних, підтримка моделі безпеки щодо виключення можливих побічних ефектів і т.д. Теоретично не існує принципових об-

межень, які заважали б встановлювати відповідність між інтерфейсом і реалізацією динамічно (деякі моделі, наприклад CORBA, підтримують концепцію динамічних інтерфейсів, але вона недостатньо формалізована). Більш того, довільна сукупність методів реалізації формально може визначити деякий інтерфейс, якщо йому надати унікальне ім'я.

Це зауваження показує, що в найбільш загальному випадку компонент може містити реалізації з більшою функціональністю, ніж того вимагають явно визначені та описані інтерфейси. А це, у свою чергу, означає, що реалізації компонента можуть носити надлишковий характер, тобто до нього може додаватися нова функціональність з наступним визначенням для неї інтерфейсу. Зворотне зауваження не вірне. Якщо до компонента додається новий інтерфейс, то для нього обов'язково повинна існувати реалізація. У цьому й полягає один з наслідків обмеження цілісності (5).

Серед множини компонентів існує особливий компонент, для якого **CInt** = ∅ і **CImp** = ∅, тобто множини інтерфейсів і реалізацій порожні. Назвемо його нульовим компонентом, або шаблоном компонента, і позначимо

$$\mathbf{TComp = (Template, \varnothing, CFact, \varnothing, CServ)}. \quad (6)$$

Умова цілісності компонента (5) для шаблона виконується. Множина вхідних інтерфейсів є порожньою і незалежно від наявності чи відсутності реалізацій вираження (5) має істинне значення.

Варто відзначити, що подання цього компонента фактично є основою засобів автоматизації створення компонентів, при застосуванні яких розробник може зосередитися на інтерфейсах і функціональності майбутнього компонента, а функції керування екземплярами, взаємодії із системними сервісами, оформлення компонента як цілісної структури включаються автоматично за допомогою інструментальних засобів.

Нехай **OldComp** визначає базовий компонент до застосування операцій, а **NewComp** відповідає отриманому компоненту після виконання операцій і

$$\mathbf{OldComp = (OldCName, OldCInt, CFact, OldCImp, CServ)},$$

$$\mathbf{NewComp = (NewCName, NewCInt, CFact, NewCImp, CServ)}.$$

Визначимо операцію додавання нової реалізації. Особливість цієї операції полягає в тому, що в результаті множина інтерфейсів компонента може розширитися за рахунок додавання нових вихідних інтерфейсів, якщо реалізація, що додається, вимагає додаткової функціональності, що надається іншими компонентами. Позначимо додаткову множину вихідних інтерфейсів $\mathbf{NewCIntO^s = \{NewCIntO^{sq}\}}$. В окремому випадку $\mathbf{NewCIntO^s = \varnothing}$, якщо додаткова функціональність для реалізації, що додається, не потрібна.

Вище відзначалося, що існують два різновиди цієї операції: додавання операції для існуючого інтерфейсу і нового, який ще формально не визначений. Позначимо першу як **AddOImp** з формою запису

$$\mathbf{NewComp = AddOImp(OldComp, NewCImp^s, NewCIntO^s)} \quad (7)$$

і семантикою

$$\mathbf{NewCInt = OldCInt \cup NewCIntO^s},$$

$$\mathbf{NewCImp = OldCImp \cup \{NewCImp^s\}},$$

$$\mathbf{(\exists OldCInt^t \in OldCIntI)\ Provide(OldCInt^t) \subseteq \subseteq NewCImp^s},$$

де $\mathbf{NewCImp^s}$ — реалізація, що додається;

OldCIntI — множина вхідних інтерфейсів з **OldCInt**.

Умова цілісності компонента (5) виконується автоматично, тому що множина вхідних інтерфейсів залишається незмінною і з цілісності старого компонента випливає цілісність нового.

За визначенням семантики операція **AddOImp** є асоціативною і комутативною операцією. Доказ цих фактів є наслідком з аналізу множин інтерфейсів і реалізацій, що входять до

складу відповідних компонентів. Використовуються асоціативні і комутативні властивості операцій над множинами.

Другий різновид операції додавання реалізації позначимо як **AddNImp** з формою запису

$$\mathbf{NewComp = AddNImp(OldComp, NewCImp^s, NewCIntO^s)} \quad (8)$$

і семантикою

$$\mathbf{NewCInt = OldCInt \cup NewCIntO^s},$$
$$\mathbf{NewCImp = OldCImp \cup \{ NewCImp^s\}},$$

де $\mathbf{NewCImp^s}$ — реалізація, що додається.

Як і для попередньої операції, цілісність компонента виконується. Також аналогічно доводиться й асоціативність і комутативність цієї операції.

Визначимо операцію заміщення існуючої реалізації новою **ReplImp** як

$$\mathbf{NewComp = ReplImp(OldComp, NewCImp^s, NewCIntO^s, OldCImp^r, OldCIntO^r)} \quad (9)$$

з наступною семантикою. Якщо справедливо, що

$$\mathbf{(\forall OldCInt^t \in OldCIntI) \&}$$
$$\mathbf{\& (Provide(OldCInt^t) \subseteq OldCImp^r) =>}$$
$$\mathbf{=> (Provide(OldCInt^t) \subseteq NewCImp^s) V}$$
$$\mathbf{V ((\exists OldCImp^j \in (OldCImp \setminus \{OldCImp^r\})) \&}$$
$$\mathbf{\& Provide(OldCInt^t) \subseteq OldCImp^j)},$$

то

$$\mathbf{NewCInt = OldCInt \cup NewCIntO^s \setminus \setminus OldCIntO^r}, \quad (10)$$
$$\mathbf{NewCImp = OldCImp \cup \{NewCImp^s\} \setminus \setminus \{OldCImp^r\}},$$

де $\mathbf{NewCImp^s}$ — реалізація, що додається;

$\mathbf{NewCIntO^s}$ — множина додаткових вихідних інтерфейсів для реалізації, що додається;

$\mathbf{OldCImp^r}$ — реалізація, що заміщується;

$\mathbf{OldCIntO^r}$ — множина вихідних інтерфейсів, пов'язаних з реалізацією, що заміщується.

Доведемо наступну лему.

Лема 1. Операція заміщення реалізації (9) із семантикою (10) зберігає умову цілісності компонента.

Для будь-якого вхідного інтерфейсу отриманого компонента, крім інтерфейсів, які відповідають реалізації $\mathbf{OldCImp^r}$, що заміщується, умова цілісності виконується. Для інтерфейсів, що відповідають реалізації $\mathbf{OldCImp^r}$, згідно передумови операції справедливо, що або $\mathbf{Provide(OldCInt^t) \subseteq NewCImp^s}$, або існує інша відповідна реалізація базового компонента. Об'єднуючи ці два випадки, отримуємо, що для будь-якого вхідного інтерфейсу нового компонента умова цілісності виконується.

Операція розширення існуючої реалізації семантично еквівалентна операції заміщення, де видаляється стара реалізація, а нова, що є розширенням старої, додається. Тому немає необхідності вводити окрему операцію.

Розглянемо операції додавання інтерфейсів. Як вже було відзначено, вихідний інтерфейс може додаватися, тільки якщо заміщається існуюча або додається нова реалізація. У противному випадку поняття додавання нового вихідного інтерфейсу не має сенсу (нема тієї реалізації, у якій міститься звертання до нових додаткових компонентів). Отже операція додавання вихідного інтерфейсу завжди є складовою операції додавання реалізації і як самостійна операція не визначається.

Визначимо операцію додавання нового вхідного інтерфейсу **AddInt** з формою запису

$$\mathbf{NewComp = AddInt(OldComp, NewCIntI^q)} \quad (11)$$

і з наступною семантикою. Якщо справедлива умова, що

$$\mathbf{(\exists OldCImp^s \in OldCImp) \&}$$
$$\mathbf{\& (Provide(NewCIntI^q) \subseteq OldCImp^s)},$$

то

$$\mathbf{NewCInt = OldCInt \cup \{NewCIntI^q\}}, \quad (13)$$
$$\mathbf{NewCImp = OldCImp},$$

де $\mathbf{NewCIntI^q}$ — новий інтерфейс.

Як видно із семантики, операція має умовний характер, тобто є частковою операцією. Доведемо наступну лему.

Лема 2. Операція додавання інтерфейсу (11) із семантикою (12) зберігає умову цілісності компонента.

Нехай виконується умова (5) для базового компонента, тобто для кожного з вхідних інтерфейсів існує відповідна реалізація. Передумова (12) вимагає існування реалізації і для нового інтерфейсу. Тому в результаті розширення множини вхідних інтерфейсів вираження (5) теж істинно для нового компонента і цілісність зберігається.

Розглянемо операцію розширення існуючого інтерфейсу. На відміну від операції розширення існуючої реалізації, для якої немає обов'язкової вимоги її збереження у структурі компонента, всі існуючі вхідні інтерфейси повинні зберігатися. Ця вимога випливає з умов допустимості методів рефакторінгу, що розглянуті вище. Тому суть операції зводиться до додавання розширеного інтерфейсу як нового із семантикою, аналогічною попередній операції.

Розглянемо операцію розширення інтерфейсу **CFact**. Ця операція носить комплексний характер, оскільки додаткові методи, що входять у цей інтерфейс, вимагають реалізації з боку контейнера. Тому реалізація цієї операції пов'язана з існуванням нового типу контейнера. Отже, відповідно до класифікації методів рефакторінгу, наведених вище, операція розширення інтерфейсу керування екземплярами компонентів відноситься до розширеної класифікації і її розгляд виходить за рамки цієї статті.

Аналогічні судження справедливі і для аналізу операції розширення інтерфейсу системних сервісів. У цьому випадку необхідною умовою є реалізація додаткових методів з боку компонентного середовища, а сама операція відноситься до розширеної класифікації методів рефакторінгу.

Таким чином, наведений вище аналіз дозволяє визначити внутрішню компонентну алгебру.

Нехай $\mathbf{CSet = \{Comp_n\}}$ — множина компонентів, кожен з яких описується моделлю (1). Множина операцій

$$\mathbf{Refac = \{ AddOImp, AddNImp, ReplImp, AddInt\}}, \quad (13)$$

визначається виразами (7) – (12). Тоді пара **(CSet, Refac)** визначає внутрішню компонентну алгебру, що відповідає припустимим методам рефакторінгу.

Порівняльний аналіз рефакторінгу об'єктно-орієнтованого і компонентного програмування

Розглянувши сутність рефакторінгу треба зауважити, що він має багато спільного з реінженерією програмних систем. Під реінженерією розуміється сукупність процесів, націлених на модифікацію функціональних, структурних, якісних та інших показників існуючих програмних систем з метою забезпечення відповідності новим умовам і вимогам функціонування. Головні відмінності між двома напрямками полягають у цілях та умовах застосування цих процесів.

Об'єктами реінженерії є вже існуючі та деякий час функціонуючі системи, які перестали відповідати певним умовам та вимогам на сучасному етапі. Це, наприклад, може бути необхідність у реалізації нової архітектури, у переході на нову платформу чи операційну систему та ін. Тобто реінженерія має справу з об'єктами, які треба замінити. Для них, зокрема, існують якісні або кількісні оцінки деяких показників, необхідні зміни значень яких і визначають напрямки процесів реінженерії.

На противагу цьому рефакторінг націлений на створення нових об'єктів на базі існуючих. І старі і нові об'єкти продовжують своє застосування. Наприклад, компоненти, які входять до складу певного програмного продукту, підтримують реалізацію функціональності для кількох версій цього продукту. Тому рефакторінг частіше застосовується як процес зміни значень показників деякого об'єкта без заміни умов функціонування самого об'єкта.

З таких позицій проведемо порівняльний аналіз рефакторінгу компонентів і рефакторінгу для об'єктно-орієнтованого програмування (ООП). Проте ще раз відмітимо, що процес заміни однієї мови ООП на іншу не є рефакторінгом, а відноситься до реінженерії. Аналогічно заміна однієї компонентної моделі іншою також є процесом реінженерії.

Нижче наведені основні результати відповідного порівняльного аналізу.

1. Мета рефакторінгу. Ціль ООП-рефакторінгу — поліпшення структурних характеристик і якісних показників об'єктно-орієнтованих (ОО) програм. Ціль компонентного рефакторінгу — розширення функціональних та інших характеристик компонентів, що найбільш повно задовольняють вимогам результатів проектування компонентної програми.

2. Об'єкти рефакторінгу. Об'єктами ООП-рефакторінгу є ОО-класи, а для компонентного програмування — готові компоненти. У зв'язку з цим ООП-рефакторінг має великі можливості і більш різноманітний, тому що операції проводяться над вхідними текстами класів, з яких надалі будується ОО-програма. На противагу цьому компонентний рефакторінг більш обмежений у своєму розмаїтті і носить більш формальний та залежний від певних умов характер.

3. Етапи застосування рефакторінгу. ООП-рефакторінг використовує вхідні тексти класів і тому застосовується на більш ранніх етапах життєвого циклу, ніж компонентний рефакторінг. Це дозволяє використовувати його на етапі розробки самих компонентів, якщо їхні реалізації будуються відповідно до ОО-підходу.

4. Середовище рефакторінгу. ООП-рефакторінг проводиться для середовища конкретної мови програмування та системи програмування. Компонентний рефакторінг не прив'язаний до конкретного середовища програмування, тому що самі компоненти можуть розроблятися на різних мовах програмування.

5. Класифікація методів рефакторінгу. Методи ООП-рефакторінгу класифікуються відповідно до елементів ООП (змінні, атрибути, методи і т.д.) і шаблонів проектування. Методи компонентного рефакторінгу класифікуються відповідно до структурних об'єктів компонентного програмування і, зокрема, структурних елементів самого компонента.

6. Інструментальні засоби рефакторінгу. Базові інструментальні засоби ООП-рефакторінгу — це засоби обробки вхідних текстів. Існують формальні методи рефакторінгу, які застосовуються для моделей ОО-класів. Базовими інструментальними засобами компонентного рефакторінгу є ті ж самі засоби, які застосовуються для створення компонентів.

Висновки

У статті описаний процес побудови алгебраїчної моделі рефакторінгу компонентів. Послідовно розглянуті основні поняття рефакторінгу компонентів, класифікація методів, припустимі операції рефакторінгу, принципи побудови внутрішньої компонентної алгебри. Основним результатом є алгебраїчна модель рефакторінгу компонентів, формальне визначення базових операцій, умови цілісності компонентів, що отримані із застосуванням визначених операцій.

Відмінною рисою пропонованого підходу є його цілісність і взаємозв'язок з іншими методами і моделями компонентного програмування. Усі концепції, термінологія, математичні методи ув'язані в єдину схему з викладом причинно-наслідкових зв'язків. Отримані результати створюють необхідний базис для формування практичних методик і методологій рефакторінгу компонентів, а також розробки відповідних інструментальних засобів.

Наведено короткий порівняльний аналіз основних характеристик рефакторінгу для компонентного й об'єктно-орієнтованого програмування.

1. *Грищенко В.Н., Лаврищева Е.М.* Компонентно-орентированное программирование. Состояние, направления и перспективы развития. // Пробл. программирования. — 2002. — № 1—2. — С. 80—90.
2. *Грищенко В.Н., Лаврищева Е.М.* Методы и средства компонентного программирования // Кибернетика и системный анализ. — 2003. — № 1. — С. 39—55.
3. *Грищенко В.Н.* Формальные модели компонентного программирования // Пробл. программирования. — 2003. — № 2. — С. 42—57.
4. *Грищенко В.Н.* Систематизований підхід до визначення програмних компонентів // Там же. — 2001. — № 3—4. — С. 23—30.
5. *Crnkovic I., Hnich B., Jonsson T., Kiziltan Z.* Specification, Implementation and Deployment of COMPONENTS // Comm. ACM. — 2002. — Oct. — P.35—40.
6. *Фаулер М.* Рефакторинг: улучшение соответствующего кода. — СПб.: Символ-Плюс, 2003. — 432 с.
7. *Приемы* объектно-ориентированного проектирования. Патерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. — СПб: Питер, 2001. — 368 с.
8. *Грищенко В.Н.* Особливості компонентно-оріентированної розробки програмного забезпечення // Пробл. программирования. — 2001. — № 3—4. — С.75—92.
9. *Сигел Дж.* CORBA 3. — Москва: Малип, 2002. — 412 с.
10. *Roman E., Ambler S., Jewell T.* Mastering Enterprise JavaBeans. — New York: Wiley Comp. Publish., 2002. — 670 с.
11. *Цимбал А.А, Аншина М.Л.* Технологии создания распределенных систем: Для профессионалов. — СПб.: Питер, 2003. — 576 с.
12. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft/COM и Java/RMI. — М.: Мир, 2002. — 510 с.

Отримано 23.10.03

Про автора

Грищенко Володимир Миколайович,
старший науковий співробітник

Місце роботи автора:
Інститут програмних систем НАН України
просп. Академіка Глушкова, 40,
Київ-187, 03680, Україна
Тел. (044) 266 3470, 266 4286