УДК 004.4'242

М.М. Глибовець, В.М. Федорченко

ІН'ЄКЦІЯ ФУНКЦІОНАЛЬНИХ ЗАЛЕЖНОСТЕЙ У КОНТЕЙНЕРІ ІНВЕРСІЇ КЕРУВАННЯ

У статті розглянуто використання функціональних інтерфейсів для зменшення зв'язності в компонентних ОО-системах та підвищення рівня повторного використання компонентів. Запропоновано і реалізовано механізм ін'єкції функціональних залежностей між компонентами в контейнерах інверсії керування з підтримкою автоматичного узгодження сигнатур.

Вступ

Залежність між компонентами є важливою характеристикою компонентної системи, що впливає на важливі параметри програмного коду: гнучкість, ефективність супроводу та можливість повторного використання. Компонент – це блок, що має здатність до композиції з визначеними інтерфейсами взаємодії та явно описаним контекстом залежностей [1]. В цій роботі під компонентом будемо розуміти клас (або множину класів) з чітко визначеними інтерфейсами взаємодії та явною декларацією залежностей від інших компонентів.

Залежність між компонентами визначається типом залежного компонента та типом зв'язності. Останній визначає міру взаємодії між компонентами. Традиційно виділяють такі типи зв'язності: загальна зв'язність шляхом прямого доступу до спільних даних (змінних), зв'язність через наслідування та зв'язність через параметри [2]. Для об'єктно-орієнтованих компонентів (ООК) характерною є зв'язність компонентів через параметри, тобто коли компоненти взаємодіють шляхом виклику методів один одного.

Чим менша зв'язність між компонентами, тим більш гнучкою та стабільною є компонентна система по відношенню як до змін у самих компонентах, так і їх конфігурації між собою. Низький рівень зв'язності зменшує кількість помилок, дозволяє легше змінювати компоненти і як наслідок, підвищує загальну якість програмного забезпечення [3].

Наразі розроблено велику кількість формальних способів визначення міри зв'язності в об'єктно-орієнтованих (ОО) системах, але більшість з них сформульовані виключно на теоретичних засадах, не враховують емпіричні та експериментальні моделі програмної інженерії і не мають практичного обґрунтування [4]. Як наслідок, використання запропонованих формальних метрик для обрахування зменшення *декларативного* рівня зв'язності (що має практичний та економічно доцільний ефект) фактично неможливо. Наприклад, практично обґрунтований спосіб зменшення декларативної зв'язності компонентів системи через використання ін'єкції залежностей фактично не змінює формально визначених метрик зв'язності (CBO, RFC, LCOM) [5]. Дійсно, ці метрики оперують низькорівневими параметрами (факт виклику методів та типів параметрів), і не враховують архітектурних аспектів (дизайну) організації цих зв'язків.

Поняття слабкої зв'язності є фундаментальним при проектуванні програмних систем [6] і має суттєвий вплив на сучасну архітектуру об'єктно-орієнтованого програмного забезпечення [7, 8]. Одним з найуживаніших практичних способів зменшення зв'язності між об'єктами є використання принципу інверсії залежностей (dependency inversion principle [9]). За цим принципом, високорівневі модулі не мають залежати від низькорівневих і будь-які модулі мають залежати від абстракцій. Але абстракції не мають залежати від деталей реалізації; навпаки, деталі мають визначатися абстракціями.

Принцип інверсії залежностей знайшов відтворення у багатьох відомих шаблонах проектування: абстрактна фабрика, адаптер, локатор сервісу, ін'єкція залежностей та інші. В загальному випа-

дку, розрив прямої залежності між класами A і B можливий шляхом абстрагування залежності через інтерфейс взаємодії I, та виділення ще одного компонента C для інкапсуляції зв'язування конкретних екземплярів класів A та B. Останнє дозволяє зменшити зв'язності між конкретними компонентами A та B через формування непрямої залежності через інтерфейс I. Компоненти залишаються зв'язаними через залежність від спільного інтерфейсу [10].

В цій роботі визначаються залежності між об'єктно-орієнтованими компонентами через *функціональні інтерфейси* з метою зменшення декларативної зв'язності між ними. Запропоновано механізм ін'єкції таких функціональних залежностей в контейнері інверсії керування, реалізація якого передбачає автоматичне узгодження типів та сигнатур під час ін'єкції.

Контейнер інверсії керування

Одним із розповсюджених варіантів реалізації шаблона ін'єкції залежностей є контейнер інверсії керування (inversion of control container) [11]. Контейнер – це компонент, який відповідає за створення та визначення залежностей об'єктно-орієнтованого компонента через посилання на інші компоненти контейнера. IoC-контейнер не потребує, щоб компоненти мали будь-яку залежність від нього (тому такі контейнери ще називають легкими).

Контейнер інверсії керування є фабрикою, що може створювати об'єкти різних типів відповідно до зовнішньої конфігурації; при цьому така фабрика є повноцінним компонентним контейнером, оскільки також визначає і життєвий цикл компонентів: створення, визначення взаємодій та знищення.

Зауважимо, що саме використання IoC-контейнера не призводить до зменшення залежностей між компонентами. Разом з тим, легкість зміни конфігурації контейнера та акцент на абстрактному визначенні залежностей компонента (через інтерфейси) дозволяють зменшити загальний рівень зв'язності системи, а також дозволяють ефективно реорганізовувати компоненти з метою подальшої декомпозиції та подрібнення компонентів (що може призводити до зменшення залежностей та дублювання коду). Формальний опис конфігурації компонентів (у вигляді XML) дозволяє ефективно використовувати IoC-контейнер для складального програмування [12], а також як інфраструктуру для модель-орієнтованого програмування [13].

Розглянемо приклад, яким чином реорганізація компонентів в IoC-контейнері може призвести до зменшення фактичної залежності між компонентами A і B, які вже мають зв'язок через інтерфейс I. Нехай інтерфейс I містить множину абстрактних методів M_I, а компонент A використовує лише деяку підмножину методів $M_I^1 \subset M_I$. Декларативна залежність між компонентами A та B може бути зменшена, оскільки в контексті конкретної взаємодії між цими компонентами інтерфейс I є надлишковим; для цього треба виділити новий інтерфейс I_1 (що містить всі необхідні для A методи M_I^1), і змінити залежність компонента A з I на I_1. У відповідності до шаблону проектування адаптер створимо новий компонент C, який реалізує інтерфейс I_1 та має залежність від компонента B через інтерфейс I. Таким чином, міра декларативної залежності між компонентами A і B зменшилася, оскільки вони більше не пов'язані спільним інтерфейсом I. Роль контейнера інверсії керування полягає у ізоляції всіх інших компонентів системи від наслідків такої реорганізації: ні компонент A, ні компонент B не змінили свої публічні контракти; зміна зв'язку між ними потребує лише локальної зміни конфігурації контейнера.

Зазначимо, що у вищенаведеному прикладі декларативний зв'язок між A та B опосередковано існує через компонент C, який пов'язаний з обома інтерфейсами I та I_1. Подальше використання різноманітних шаблонів об'єктно-орієнтованого дизайну може збільшити кількість ланок абстракції, але не може повністю ізолювати компонент A від компоненту B.

Функціональні інтерфейси

Функціональний інтерфейс – це інтерфейс, який визначає тільки один абстрактний метод (Single Abstract Method interface). Важливою властивістю SAM-інтерфейсів є концептуальна можливість їх реалізації через посилання на метод об'єкту, інший функціональний інтерфейс з сумісною сигнатурою або лямбда-вираз. Функціональні інтерфейси дозволяють гармонійно інтегрувати шаблони функціонального програмування в об'єктно-орієнтованих мовах програмування [14].

Інваріантність функціональних інтерфейсів дозволяє у деяких випадках повністю розірвати декларативну залежність між компонентами, зберігаючи фактичну здатність цих компонентів до взаємодії. Це справджується для багатьох відомих шаблонів проектування: абстрактна фабрика (інтерфейс фабрики створення об'єкта можна звести до функціонального, тому реалізація фабрики може бути повністю незалежною від компонентів, що її використовують), відвідувач (інтерфейс відвідувача можна звести до функціонального з одним методом `visit`), команда (інтерфейс команди є функціональним з одним методом `execute`, реалізації різних команд можуть бути декларативно незалежними від компонентів обробки команд) тощо. В корпоративних системах функціональні інтерфейси зустрічаються в компонентах бізнес-правил, доступу до даних, операцій тощо.

Розглянемо підхід до повного розриву декларативної залежності. Компонент A залежить від компонента B через інтерфейс I, що складається з множини абстрактних методів $I = \{m_1,...,m_n\}$. Завжди можлива заміна залежність від інтерфейсу I на множину інтерфейсів $\{I_1,...,I_k\}$, яка повністю або частково складається з функціональних інтерфейсів. Дійсно, в загальному випадку:

$$I = \{m_1,...,m_n\} = \bigcup_{k=1}^{n} \{m_k\}.$$

Як наслідок, компонент A може декларувати залежність від локальних функціональних інтерфейсів I_k, а компонент B – визначати сервіси за допомогою інших локальних інтерфейсів або у вигляді публічних методів; компоненти A та B наразі не мають спільного інтерфейсу, що їх пов'язує, оскільки при зв'язуванні компонентів A та B можливе автоматичне узгодження функціональних інтерфейсів контейнером інверсії керування.

Практична доцільність такого подрібнення інтерфейсу I з метою розриву декларативної залежності визначається характером взаємодії та контексту методів $m_1,...,m_n$. Наш досвід розробки корпоративних систем свідчить, що у багатьох випадках таке подрібнення інтерфейсів призводить до інкапсуляції бізнес-логіки у окремих компонентах, що значно спрощує їх підтримку та модифікацію у майбутньому. Виділення таких атомарних компонентів, які можуть взаємодіяти через функціональні інтерфейси, сприяє їх повторному використанню та значно підвищує їх здатність до композиції [15].

Ін'єкція функціональних залежностей

Сучасні ОО-мови мають засоби підтримки функціональних інтерфейсів. Наприклад, в очікуваній Java 8 компілятор автоматично має узгоджувати 2 різні SAM-інтерфейси, якщо сигнатури методів збігаються за кількістю і порядком параметрів, типи вихідних параметрів коваріантні, а типи вхідних – контраваріантні. В MS.NET не підтримується автоматичне узгодження SAM-інтерфейсів, але є особливий тип (делегат), який визначається сигнатурою метода і використовується для декларації функціональної залежності.

Розглянемо технологічний аспект ін'єкції функціональних залежностей для платформи .NET. Оскільки є дві різні форми декларації таких залежностей (SAM-інтерфейс і делегат), постає проблема узгодження типів при ін'єкції у контейнері інверсії керування. Маємо чотири можливі випадки: делегат → делегат, делегат → SAM-інтерфейс, SAM-інтерфейс → делегат, SAM-інтерфейс → SAM-інтерфейс. Оскільки делегат і SAM-інтерфейс визна-

чаються сигнатурою метода, фактично потрібно реалізувати такі перетворення: метод → делегат, метод → SAM-інтерфейс.

Середовище .NET дозволяє створювати делегат потрібного типу за методом об'єкта, що має сумісну сигнатуру:

```
var toDelegate =
Delegate.CreateDelegate(toType,
fromObject, fromMethod, false);
```

Для сумісності сигнатур мають задовольнятися умови коваріантності для вихідних і контраваріантності для вихідних параметрів [16]. Більш складним є випадок ін'єкції SAM-інтерфейсу, оскільки у цьому випадку необхідно якимось чином реалізувати зазначений інтерфейс. В рамках технологічних можливостей платформи .NET реалізувати автоматичне перетворення метод → SAM-інтерфейс можливо декількома способами.

Універсальним є варіант, коли програміст заздалегідь підготує компонент-конвертор та проксі-імплементацію для функціонального інтерфейсу у відповідності до компонентної моделі .NET:

```
[TypeConverter(typeof(PermissionCheck
erConverter))]
public interface IPermissionChecker {
 bool Check(int accountId, string
operation, object context);
}
public class PermissionCheckerProxy :
IPermissionChecker {
 Delegate F;
 public
PermissionCheckerProxy(Delegate f) {
F = f; }
 public bool Check(int accountId,
string operation, object context) {
  return (bool)F.DynamicInvoke(new
object[]
{accountId,operation,context});
 }
}
public class
PermissionCheckerConverter :
TypeConverter {
 public override object
ConvertFrom(ITypeDescriptorContext
context,

CultureInfo culture, object value) {
  if (value is Delegate)
   return new PermissionCheckerProxy(
(Delegate)value );
  return
base.ConvertFrom(context,culture,valu
e);
 }
}
```

Для узгодження такого SAM-інтерфейсу треба скористатись загальною інфраструктою перетворення типів компонентної моделі .NET:

```
var toTypeConv =
TypeDescriptor.GetConverter(typeof(IP
ermissionChecker));
if
(toTypeConv.CanConvertFrom(delegateTy
pe)) {
 return
toTypeConv.ConvertFrom(delegate);
}
```

Суттєвим недоліком цього варіанту є створення спеціального класу-конвертора та проксі класу для кожного функціонального інтерфейсу.

У .NET існує спосіб реалізації будь-якого інтерфейсу через наслідування особливого класу `System.Runtime.Remoting.Proxies.RealProxy`. Цей клас призначений для організації віддалених викликів .NET Remoting і дозволяє обробляти виклики методів інтерфейсу у вигляді повідомлень. Подібний механізм можна використати для локальних викликів, і таким чином визначити універсальний проксі для будь-якого SAM-інтерфейсу:

```
class InterfaceAdapter :
System.Runtime.Remoting.Proxies.RealP
roxy {
 Delegate F;
 public override IMessage
Invoke(IMessage m) {
  if (m is IMethodCallMessage) {
   var methodCall =
(IMethodCallMessage)m;
   var response = F.DynamicInvoke(
methodCall.Args );
   return new ReturnMessage(response,
null, 0, null, methodCall);
  }
  throw new
NotImplementedException();
 }
}
```

Недоліком такої реалізації є суттєві накладні витрати, що спричиняє RealProxy.

Результати тестування свідчать, що такий проксі-об'єкт працює у 7 разів повільніше за попередній варіант. Тим не менше, для переважної більшості застосувань (коли залежність від SAM-інтерфейсу викликається обмежену кількість разів) використання RealProxy є цілком виправданим.

Для збереження можливості впливати на реалізацію проксі класу ми пропонуємо комбінований варіант. Спочатку перевіряється, чи має інтерфейс власний компонент-конвертор для створення проксі-об'єкта за делегатом. Якщо конвертор явно не визначений, використовується універсальний проксі на основі RealProxy.

Узгодження сигнатур функціональних інтерфейсів

Ін'єкція функціональної залежності (у вигляді SAM-інтерфейсу чи типу делегата) передбачає використання повністю сумісної сигнатури. Наприклад, нехай є деякий інтерфейс:

```
public interface ITest {
 string DoSomething(string s,
bool b);
}
```

Залежність від `ITest` може бути визначена за делегатом з типом `Func<string,bool,string>`, але спроба ін'єкції делегата типу `Func<string,bool,int>` призведе до помилки, оскільки результат `int` не може бути приведений до `string`. У таких випадках розробник вимушений створювати надлишкові методи-синхронізатори, наприклад:

```
public string DoSomethingStr(string
s, bool b) {
 return origDelegate(s,b).ToString();
}
```

Подібні синхронізатори у тривіальних випадках перетворень типів призводять до великої кількості надлишкового коду та ускладнюють конфігурацію компонентів з використанням ін'єкції функціональних залежностей. Цього можна уникнути, якщо при узгодженні таких залежностей перевіряти контраваріантність типів аргументів та коваріантність типу результату, та виконувати додаткове перетворення при потребі. У випадку ін'єкції SAM-інтерфейса у `InterfaceAdapter` для цього треба дещо ускладнити обробку виклику метода:

```
public override IMessage
Invoke(IMessage m) {
 if (m is IMethodCallMessage) {
  var methodCall =
(IMethodCallMessage)m;
  var args =
(object[])methodCall.Args.Clone();
  // check args contravariance
  var mParams =
Method.GetParameters();
  for (int i = 0; i < args.Length;
i++) {
   if (args[i] != null) {
    if (mParams.Length > i &&

!mParams[i].ParameterType.IsAssignabl
eFrom(args[i].GetType())) {
     args[i] =
ConvertManager.ChangeType(args[i],
mParams[i].ParameterType);
    }
   }
  }
  var response =
Method.Invoke(Target, args);
  if (response != null &&
      InterfaceMethod.ReturnType !=
typeof(void) &&

!InterfaceMethod.ReturnType.IsAssigna
bleFrom(response.GetType())) {
   response =
ConvertManager.ChangeType(response,
InterfaceMethod.ReturnType);
  }
  return new ReturnMessage(response,
null, 0, null, methodCall);
 }
 throw new NotImplementedException();
}
```

При узгодженні сигнатур делегатів ситуація дещо складніша, оскільки як аргумент `Delegate.CreateDelegate` очікує метод з сумісною сигнатурою. Якщо сигнатура не є сумісною, потрібне використання особливого проксі-метода сумісної сигнатури.

Розглянемо технічні особливості реалізації такого проксі-метода. Аргументи можуть мати тип `object`, оскільки він є контраваріантним до будь-якого типу в .NET. Підтримку різної кількості аргумен-

тів забезпечимо визначенням N-методів, що мають $0..N$ аргументів (для практичного застосування $N=15$ виглядає цілком прийнятним обмеженням). Для забезпечення коваріантного типу результату, ми скористаємось шаблонними методами:

```
class DelegateAdapter {
 object Target;
 MethodInfo Method;
 protected object Invoke(params
object[] args) {
  var mParams =
Method.GetParameters();
  for (int i = 0; i < args.Length;
i++) {
   if (args[i]!=null) {
    if (mParams.Length > i &&

!mParams[i].ParameterType.IsAssignabl
eFrom(args[i].GetType())) {
     args[i] =
ConvertManager.ChangeType(args[i],
mParams[i].ParameterType);
    }
   }
  }
  return Method.Invoke(Target, args);
}
public T Invoke<T>() {
 return
(T)ConvertManager.ChangeType(Invoke()
, typeof(T));
}
public T Invoke<T>(object arg1) {
 return
(T)ConvertManager.ChangeType(Invoke(a
rg1), typeof(T));
}
public T Invoke<T>(object arg1,
object arg2) {
 return
(T)ConvertManager.ChangeType(Invoke(a
rg1,arg2), typeof(T));
 }
 /* ... */
}
```

Для створення делегату потрібного типу при ін'єкції залежності маємо знайти шаблонний метод `Invoke` з потрібною кількістю аргументів та сконструювати метод з необхідним типом результату:

```
var adapter = new
DelegateAdapter(fromObject,
fromMethod);
foreach (var adapterMethod in
adapter.GetType().GetMethods()) {
 if (adapterMethod.Name == "Invoke"
&&
adapterMethod.GetParameters().Length
== toMethodParamCount) {
  var resType =
toMethodInfo.ReturnType !=
typeof(void) ?

toMethodInfo.ReturnType :
typeof(object);
  var typedInvokeMethod =
adapterMethod.MakeGenericMethod(resTy
pe);
  return
Delegate.CreateDelegate(toType,
adapter, typedInvokeMethod, true);
 }
}
```

Повну реалізацію узгодження функціональних інтерфейсів та делегатів у вигляді окремого компонента можна знайти за адресою **Помилка! Неприпустимий об'єкт гіперпосилання.**. Компонент призначений для інтеграції з контейнером інверсії керування Winter4Net, але його можливо використовувати і з іншими контейнерами, наприклад, Spring.NET.

Висновки

Компонентна розробка передбачає, що програмне забезпечення збирається з бібліотек вже написаних компонентів, а процес збирання може бути в значній мірі автоматизований у вигляді фабрик програм. Програмні компоненти мають різні форми і механізми взаємодії, тому проблема узгодження їх взаємодії залишається актуальною. Вирішення цієї задачі полягає як у площині розвитку теорії об'єктного й компонентного програмування, так і у відповідній технологічній підтримці компонентних середовищ [17].

В рамках такої технологічної підтримки у цій роботі ми розглянули використання функціональних залежностей між ОО-компонентами, які дозволяють повністю абстрагувати компоненти один від одного, при збереженні їх здатності до взаємодії і композиції. Підтримка ін'єкції таких залежностей в контейнерах інверсії керування потребує особливого механізму узгодження типів, і ми розглянули технологічні аспекти такого узгодження для платформи MS.NET. Була запропонована реалізація механізму, який дозволяє авто-

матично узгоджувати делегати та SAM-інтерфейси. Узгодження можливе навіть при формально несумісних типах параметрів (при наявності формально визначених правил перетворення типів), що надзвичайно важливо для автоматичного збирання конфігурацій компонентів у фабриках програм.

1. *Szyperski* Clemens. Component Software: Beyond Object-Oriented Programming. – 2nd edition. – Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
2. *Yu Li-Guo, Ramaswamy Srini.* Component Dependency in Object-Oriented Software // Journal of Computer Science and Technology. – 2007. – Vol. 22, N 3. – P. 379–386.
3. *Arisholm Erik, Briand Lionel C., Foyen Audun.* Dynamic Coupling Measurement for Object-Oriented Software // IEEE Trans. Softw. Eng. – 2004. – August. – Vol. 30, N 8. – P. 491–506.
4. *Briand Lionel C., Daly John W., Wust Jurgen K.* A Unified Framework for Coupling Measurement in Object-Oriented Systems // IEEE Trans. Softw. Eng. – 1999. – Vol. 25, N 1. – P. 91–121.
5. *Razina Ekaterina, Janzen David.* Effects of dependency injection on maintainability // Proceedings of the 11th IASTED International Conference on Software Engineering and Applications. – SEA '07. – Anaheim, CA, USA: ACTA Press, 2007. – P. 7–12.
6. *Stevens W. P., Myers G. J., Constantine L.L.* Structured design // IBM Syst. J. – 1974. – June. – Vol. 13, N 2. – P. 115–139.
7. *Booch Grady.* Object-oriented analysis and design with applications (2nd ed.). – Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994.
8. *Coad Peter, Yourdon Edward.* Object-oriented analysis (2nd ed.). – Upper Saddle River, NJ, USA: Yourdon Press, 1991.
9. *Martin Robert C.* The Dependency Inversion Principle // C++ Report. – 1996. – May. – Vol. 8.
10. *Fowler Martin.* Reducing Coupling // IEEE Software. – 2001. – Vol. 18, N 4. – P. 102–104.
11. *Walls Craig, Breidenbach Ryan.* Spring in action. – Greenwich, CT, USA: Manning Publications Co., 2007.
12. *Федорченко В.М.* Каркас для підтримки модель-орієнтованої розробки на основі спрощеної інфраструктури трансформації XML-моделей // Наукові записки. Том 86, Комп'ютерні науки. Національний університет "Києво-Могилянська академія". – 2008. – Т. 83. – P. 61–65.
13. *Glibovets N.N., Fedorchenko V.M.* Simplified infrastructure for the transformation of XML models // Cybernetics and Sys. Anal. – 2010. – January. – Vol. 46, N 1. – P. 93–97.
14. *Kuhne Thomas.* Higher order objects in pure object-oriented languages // SIGPLAN Not. – 1994. – July. – Vol. 29, N 7. – P. 15–20.
15. *Elizondo Perla Velasco, Ndjatchi Mbe Koua Christophe.* Deriving functional interface specifications for composite components // Proceedings of the 10th international conference on Software composition. – SC'11. – Berlin, Heidelberg: Springer-Verlag, 2011. – P. 1–17.
16. *Svendsen Kasper, Birkedal Lars, Parkinson Matthew.* Verifying generics and delegates // Proceedings of the 24th European conference on Object-oriented program-ming. – ECOOP'10. – Berlin, Heidelberg: Springer-Verlag, 2010. – P. 175–199.
17. *Андон П., Лавріщева К.* Розвиток фабрик програм в інформаційному світі // Вісник НАН України. – 2010. – № 10. – P. 15–41.

Одержано 09.12.2013

Про авторів:

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

Федорченко Віталій Михайлович,
аспірант.

Місце роботи авторів:

Національний університет
«Києво-Могилянська Академія»,
04655, Київ-70,
вул. Г. Сковороди 2.
Тел. (067) 209 0740.
Факс (044) 416 4515.
E-mail: glib@ukma.kiev.ua