УДК 004.41
DOI: 10.18523/2617-3808.2023.6.57-61

Федорченко В. М.

ДЕКЛАРАТИВНЕ КОНФІГУРУВАННЯ ІНВЕРСІЇ КОНТРОЛЮ В СУЧАСНИХ .NET8 ЗАСТОСУНКАХ

З огляду на постійне підвищення складності сучасних програмних систем стає дедалі важливішим значення впровадження компонентно-орієнтованих архітектурних парадигм. Шаблони інверсії контролю (IoC) та ін'єкції залежностей (DI) відіграють ключову роль у керуванні залежностями об'єктно-орієнтованих (ОО) компонентів, зменшенні зв'язності та забезпеченні безперешкодної інтеграції компонентів. Ця стаття досліджує теоретичні основи IoC та DI, розкриваючи їх практичну реалізацію в сучасних додатках .NET 8.

Ключові слова: інверсія контролю, ін'єкція залежностей, компонент-орієнтована розробка, .NET Core, .NET8.

Абстрагування залежностей між ОО-компонентами є одним із ключових факторів, що визначають важливі характеристики програмного коду: гнучкість, ефективність супроводу та можливість повторного використання. Низький рівень зв'язності зменшує кількість помилок, дає змогу легше змінювати компоненти, проводити їх незалежне тестування і, як наслідок, підвищує загальну якість програмного забезпечення [2]. Для забезпечення низького рівня зв'язності використовують такі архітектурні шаблони, як інверсія контролю (Inversion-of-Control) та ін'єкція залежностей (Dependency Injection) [2; 6]. Сутність шаблону «інверсія керування» полягає в абстрагуванні створення та ініціалізації компонентів через делегування цих дій особливому компоненту (DI-контейнеру). Створення компонентів і визначення залежностей (dependencies resolution) у вигляді конкретних екземплярів (об'єктів) виконує такий контейнер; компонентам більше не потрібно знати, як створювати чи налаштовувати свої залежності, що робить їх більш незалежними та здатними до адаптації. Це приводить до коду, який легше тестувати, підтримувати та розширювати.

У сучасних об'єктно-орієнтованих програмних платформах подібні архітектурні рішення стали невід'ємною частиною, і Microsoft .NET не є винятком. Починаючи з .NET Core 1.0 (2016) всі стандартні компоненти платформи розроблено з урахуванням того, що життєвий цикл і залежності цих компонентів (сервісів) контролюються вбудованим DI-контейнером [4], а зв'язування та визначення реалізацій інтерфейсів повністю абстраговано від самих компонентів.

Розглянемо компонент як блок, що має здатність до композиції з визначеними інтерфейсами взаємодії та явно описаним контекстом залежностей. Такий програмний компонент може бути використаний незалежно і передбачає можливість композиції з іншими компонентами. Одна з проблем компонентної розробки полягає в тому, що вона потребує від розробників відстеження залежностей, щоб визначити компоненти, які мають стосунок до задачі, і вирішити, які компоненти та як слід повторно використовувати [7]. В контексті об'єктно-орієнтованої програми компоненти зазвичай визначаються у вигляді класу (або групи пов'язаних класів) з відповідно визначеним інтерфейсом та чітко окресленими залежностями (C#):

```
public class AComponent :
IMyComponent
{
    TextWriter Output;
    public AComponent(TextWriter
output) {
        Output = output;
    }
    public void WriteMessage(string
message) {
        output.WriteLine($»Message:
{message}»);
    }
}
```

Декларація цього компонента (сервісу) в стандартному DI-контейнері виконується шляхом виклику IServiceCollection.Add (Service Descriptor) або відповідного методу-розширення (код в методі Startup.ConfigureServices):

```
services.AddScoped<IMyComponent,MyCo
mponent>();
```

Якщо в декларації компонента немає експліцитного визначення необхідних залежностей (параметрів конструктора MyComponent), контейнер імпліцитно визначає відповідні посилання за типами цих параметрів.

Варто зауважити, що ця вбудована реалізація DI-контейнера має суттєві обмеження, які перешкоджають застосуванню деяких аспектів техніки інверсії контролю:

- не підтримується можливість визначати декілька компонентів, які мають той самий тип, що значно обмежує здатність компонентів до композиції;
- не підтримується імпліцитна ін'єкція залежностей, які визначені у вигляді публічної властивості класу (property injection);
- немає підтримки декларативного визначення компонентів та їх залежностей у вигляді JSON або XML конфігурації.

У найновішій версії платформи .NET8 (2023) з'явилась підтримка іменованих сервісів (keyed services), що дає змогу визначити декілька компонентів одного типу:

```
services.AddKeyedScoped<AComponent>(
"component1");
services.AddKeyedScoped<AComponent>(
"component2");
```

Щоб отримати посилання саме на конкретний компонент, декларація залежності має виглядати так:

```
public BComponent([FromKeyedServices
("component1")]MyComponent c)
```

Нескладно помітити суттєвий недолік: подібна декларація порушує принцип інверсії контролю, тому що реалізація (компонент) містить посилання на ідентифікатор компонента. Цьому можна запобігти через експліцитне визначення залежностей:

```
services.
AddKeyedScoped<BComponent>(srv =>
  new BComponent(srv.GetRequiredKeye
dService<AComponent>("component1")));
```

Так само можна зробити ін'єкцію властивостей, але такий опис має форму імперативного коду для визначення всіх залежностей (яких у реальних компонентів буває багато), що само по собі підвищує зв'язність в застосунку та провокує появу значної кількості шаблонного (і надлишкового) коду, який конфігурує DI-контейнер. Ці особливості стандартного контейнера сприяють ситуації, коли в реальних .NET програмах лише обмежена кількість компонентів «верхнього рівня» реєструється в контейнері, самі компоненти мають надлишкову зв'язність, а багато більш низькорівневих компонентів залишаються зв'язані жорстко і для них інверсія контролю не застосовується.

Для вирішення цих недоліків стандартного DI-контейнера пропонується реалізувати розширення у вигляді фабрики компонентів, яка буде реєструвати компоненти в контейнері та визначати їх залежності відповідно до декларативної конфігурації, що може зберігатися в стандартному (для .NET Core застосунків) файлі appsettings.json, або завантажуватись з окремого JSON файлу (або взагалі якимось чином генеруватись динамічно):

```
[
  {Type: "AComponent", Name:
"component1", Lifetime: "Scoped"},
  {Type: "AComponent", Name:
"component2"},
  {
   Type": "BComponent",
   Constructor: [{$ref: "component1"}],
   Properties: {
         ValueProp: 10,
         DependencyProp: {$ref:
                         "component1"}
    }
  },
]
```

Розглянемо основні технічні аспекти реалізації такої фабрики компонентів.

Для внутрішнього представлення опису компонентів та їх залежностей (які декларуються як параметри конструктора класу та/або публічні set-властивості) визначимо таку об'єктну модель.

Відповідно до цієї моделі весь граф залежностей компонентів може бути визначений як множина ComponentDescriptor. Залежності конкретного компонента визначаються через списки параметрів конструктора та/або ін'єкції властивостей класу компонента. Кожна така залежність може бути посиланням на інший компонент (RefDescriptor), або значенням простого типу даних (ValueDescriptor), або списком чи словником(ListDescriptortaDictionaryDescriptor), значення якого своєю чергою можуть бути описані як посилання на компоненти чи прості значення. Посилання на інший компонент може

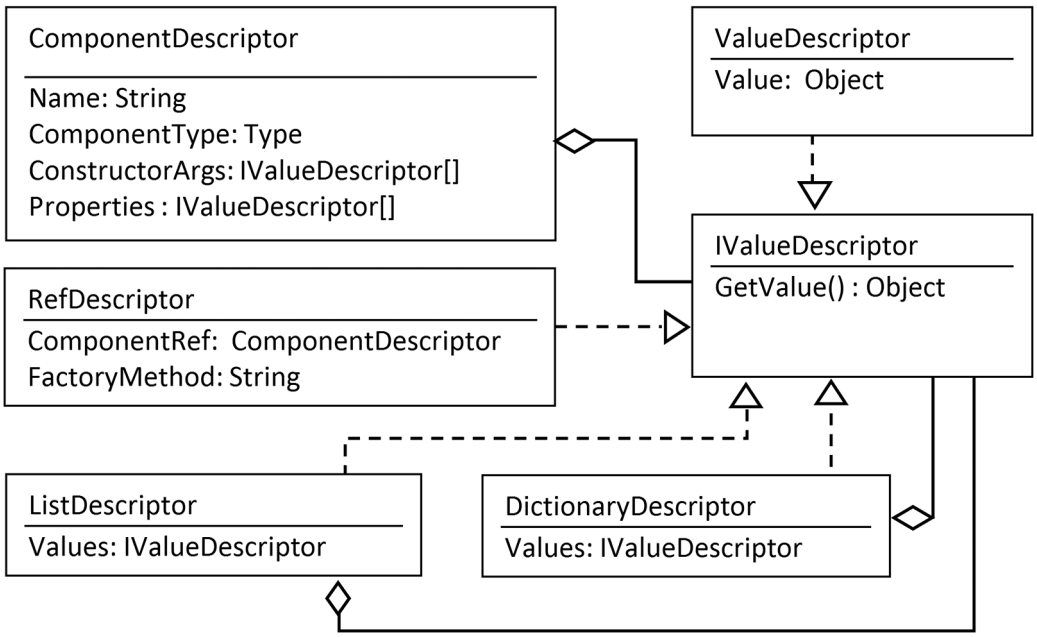

Рис. 1. Об'єктна модель опису компонентів

бути за назвою, за типом або як inline-визначення. Подібне внутрішнє представлення дає змогу абстрагувати фабрику компонентів від конкретного формату конфігурації, який може бути не тільки JSON, а й XML або YAML.

Функціональна декомпозиція розширення стандартного DI-контейнера, яке реалізує підтримку декларативної конфігурації залежностей, складатиметься з таких елементів:

- **JsonLoader**. Оскільки файл конфігурації сучасних .NET застосунків (appsettings.json) має JSON формат, для парсингу конфігурації залежностей використаємо стандартний пакет System.Text.Json. Ітеруючи елементи JsonDocument, створимо об'єктну модель графу залежностей компонентів. На цьому етапі виконуються валідація та зв'язування всіх типів компонентів, вказаних у конфігурації. Для цього призначена типова реалізація ITypeResolver, яка використовує механізм відображень (Reflection) для визначення типів (System.Type) за їх повною або частковою назвою. При цьому .NET застосунок має можливість визначити свою реалізацію ITypeResolver для підтримки скорочених назв типів; наприклад, це можуть бути списки пріорітетних областей імен (namespaces), в яких буде проводитись пошук таких скорочених назв типів.
- **ComponentFactory**. Реалізація фабрики конкретного компонента, яка створює екземпляри відповідно до опису у вигляді ComponentDescriptor. Ця процедура передбачає пошук найбільш відповідного конструктора класу компонента (якщо їх декілька), визначення експліцитних (за посиланням у конфігурації) та/або імпліцитних посилань (за типом параметра конструктора) на екземпляри інших компонентів. Оскільки життєвий цикл компонентів контролюється стандартним DI-контейнером, для отримання цих посилань на інші компоненти фабрика визначає залежність від IComponentContainer, який своєю чергою реалізується через механізм IServiceProvider стандартного DI-контейнера (ServiceProviderComponentFactory). Варто зазначити, що декларативна конфігурація конкретної ін'єкції може визначатись як inline-визначення компонента; такі компоненти не реєструються в DI-контейнері, а отже, їх екземпляри створюються безпосередньо (без використання механізму IServiceProvider).
- **IServiceCollection.LoadComponentsFrom Json**. Інтеграція фабрики компонентів зі стандартним DI-контейнером імплементується через метод розширення (extension method), який використовує JsonLoader для завантаження конфігурації у вигляді об'єктної моделі (множини ComponentDescriptor), реєструє компоненти в DI-контейнері, і як фабричний метод (який створює екземпляр компонента) використовує ComponentFactory:

```
foreach (ComponentDescriptor c in
components) {
 var factory = new
```

```
ComponentFactory(c);
 Func<IServiceProvider, object,
object> create = (srvPrv, key) =>
{
  return factory.Create(
    new ServiceProviderComponentF
actory(srvPrv),
c.ImplementationType);
 };
 var descriptor =
ServiceDescriptor.DescribeKeyed(
  c.ServiceType ?? c.
ImplementationType,
  c.Name, create, c.Lifetime);
 services.Add(descriptor);
}
```

Отже, для підключення декларативної конфігурації компонентів у будь-якому сучасному . NET застосунку достатньо додати лише один рядок коду в Startup.ConfigureServices:

```
services.LoadComponentsFromJson("com
ponents.json");
```

Повна реалізація фабрики компонентів розміщена на GitHub у вигляді бібліотеки: https://github.com/nreco/dependencyinjection.

За допомогою такого розширення шаблон «інверсія контролю» може бути застосований не тільки до прикладних компонентів, а й до самого DI-контейнера. Оскільки конфігурування залежностей повністю абстраговане від програмного коду застосунку, зміна цієї конфігурації не потребує перекомпіляції застосунку, що своєю чергою значно підвищує можливості адаптації програми до конкретного середовища виконання — наприклад, версії для різних операційних систем або різних провайдерів хмарного хостингу. Наявність такого альтернативного способу декларації компонентів створює сприятливі технічні можливості для активного використання функціональних інтерфейсів при декомпозиції [5; 8] та підтримку ін'єкцій таких функціональних залежностей з автоматичним узгодженням [1].

Сама наявність декларативної конфігурації DI-контейнера відкриває широкі можливості для генеративного програмування. Стає технічно можливим впровадження модель-орієнтованої розробки, зокрема підходу, який використовує конфігурацію DI-контейнера як кінцеву модель рівня виконання в ланцюжку трансформацій предметно-специфічних моделей [3].

Компонент-орієнтована розробка є загальноприйнятим ефективним методом зниження витрат виробництва програмних продуктів, скорочення часу виходу на ринок та підвищення якості [7]. Сучасна платформа .NET8 передбачає, що програмне забезпечення збирається з бібліотек уже написаних компонентів та надає стандартизований механізм середовища часу виконання для цих компонентів. Розглянуто практичні аспекти застосування шаблону «інверсія контролю» за умови використання стандартної реалізації DI-контейнера, визначено наявні технологічні обмеження, а також запропоновано рішення для декларативного визначення залежностей між компонентами, яке сприяє низькому рівню зв'язності між компонентами та декомпозиції програми на більш гранулярні та більш абстрактні компоненти, що також підвищує імовірність їх повторного використання.

Список літератури

1. Глибовець М. М. Ін'єкція функціональних залежностей у контейнері інверсії керування / М. М. Глибовець, В. М. Федорченко // Проблеми програмування. — 2014. — № 4. — С. 33–39.
2. Fowler M. Reducing Coupling / Martin Fowler // IEEE Software. — 2001. — Vol. 18, no. 4. — Pp. 102–104.
3. Glibovets N. N. Simplified infrastructure for the transformation of XML models / N. N. Glibovets, V. M. Fedorchenko // Cybernetics and Sys. Anal. — 2010. — Vol. 46, no. 1. — Pp. 93–97. https://doi.org/10.1007/s10559-010-9187-0.
4. Larkin K. Dependency injection in ASP.NET Core [Electronic resource] / K. Larkin, S. Smith, B. Dahler. — Mode of access: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection.
5. Lecessi R. Functional Interfaces in Java / Ralph Lecessi. — Apress, 2019.
6. Martin R. The Dependency Inversion Principle / Martin Robert C. // C++ Report. — 1996. — Vol. 8.
7. Szyperski C. Component Software: Beyond Object-Oriented Programming / Clemens Szyperski. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.
8. Velasco Elizondo P. Deriving functional interface specifications for composite components / P. Velasco Elizondo, M. K. C. Ndjatchi // Springer eBooks. — 2011. — Pp. 1–17. https://doi.org/10.1007/978-3-642-22045-6_1.

References

Fowler, M. (2001). Reducing coupling. *IEEE Software, 18* (4), 102–104. https://doi.org/10.1109/ms.2001.936226.

Glibovets, N. N., & Fedorchenko, V. M. (2010). Simplified infrastructure for the transformation of XML models. *Cybernetics and Systems Analysis, 46* (1), 93–97. https://doi.org/10.1007/s10559-010-9187-0.

Hlybovets, M. M., & Fedorchenko, V. M. (2014). Iniektsiia funktsionalnykh zalezhnostei u konteineri inversii keruvannia. *Problemy prohramuvannia*, *4*, 33–39.

Larkin, K., S. Smith, S., & B. Dahler,. B. — *Dependency injection in ASP.NET Core*. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection.

Lecessi, R. (2019). *Functional interfaces in Java: Fundamentals and Examples*. Apress.

Martin, R. (1996). The Dependency Inversion Principle. *C++ Report*, *8*.

Szyperski, C. (2002). *Component software: Beyond Object-Oriented programming.*

Velasco-Elizondo, P., & Ndjatchi, C. (2011). Deriving functional interface specifications for composite components. In *Springer eBooks* (pp. 1–17). https://doi.org/10.1007/978-3-642-22045-6_1.

V. Fedorchenko

DECLARATIVE INVERSION OF CONTROL CONFIGURATION IN MODERN .NET8 APPLICATIONS

As software systems continue to evolve in complexity and scale, the importance of adopting component-based architectural paradigms becomes increasingly evident. Inversion of Control (IoC) and Dependency Injection (DI) patterns play important role in managing OO-components dependencies, reduce coupling and enable seamless integration of components. This article explores the theoretical foundations of IoC and DI, shedding light on their practical implementation in modern .NET applications.

Standard .NET DI-container has a number of technical limitations: properties injection is not supported, dependencies definitions from keyed services partially breaks IoC principle, and finally lack of declarative way to define components and their dependencies. To address these limitations, it is proposed to implement a special extension for standard DI-container in the form of a component factory. This factory would register components within the container and define their dependencies based on declarative configuration. The configuration can be stored either in a standard (for .NET Core apps) appsettings.json file or loaded from a separate JSON file. In fact, this JSON could be even dynamically generated in specific usage scenarios.

The capability to use a declarative configuration for standard DI-container opens up broad possibilities for generative programming. It becomes technically feasible to implement a model-driven development, particularly an approach that leverages the DI-container configuration as the final (execution) model in a chain of transformations of domain-specific models.

Modern .NET8 platform assumes that software is assembled from libraries of pre-existing components that are hosted in a standard DI-container that provides suitable runtime environment for them. Practical aspects of applying the Inversion of Control pattern are examined, considering the use of a standard Dependency Injection (DI) container implementation. Existing technological limitations are defined, and a solution is proposed: implementation of declarative configuration of dependencies between components. This approach contributes to a reduced level of coupling between components and facilitates the decomposition of the program into more granular and abstract components – which increases their reusability in consequence.

Keywords: inversion of control, dependency injection, component based development, declarative IoC configuration, .net core, .NET8.

Матеріал надійшов 01.12.2023