5. Dependency Injection Container

​

Необходимо реализовать простой Dependency Injection контейнер.

Dependency Injection контейнер — это обобщенная и конфигурируемая фабрика объектов. Типы данных, объекты реализации которых может создавать DI контейнер, далее будем называть **зависимостями**.

Контейнер должен позволять регистрировать зависимости в формате: Тип интерфейса (TDependency) -> Тип реализации (TImplementation), где TDependency — любой ссылочный тип данных, а TImplementation — не абстрактный класс, совместимый с TDependency, объект которого может быть создан.

Контейнер должен быть отделен от своей конфигурации: сначала выполняется создание конфигурации и регистрация в нее зависимостей, а затем создание на ее основе контейнера. Должна обеспечиваться **валидация** конфигурации контейнера в момент создания контейнера.

// иллюстрация вышесказанного

// конкретный API регистрации/получения зависимостей на усмотрение автора

var dependencies = new DependenciesConfiguration();

dependencies.Register<IService1, Service1>();

dependencies.Register<AbstractService2, Service2>();

​

// тип зависимости может совпадать с типом реализации

// иногда это называют регистрацией "as self":

dependencies.Register<Service3, Service3>();

var provider = new DependencyProvider(dependencies);

var service1 = provider.Resolve<IService1>();

...

Внедрение зависимостей должно осуществляться через конструктор. Создание зависимостей должно выполняться **рекурсивно**, то есть если TImplementation имеет свои зависимости, а каждая из его зависимостей — свои (и т. д.), то контейнер должен создать их все:

interface IService {}

class ServiceImpl : IService

{

public ServiceImpl(IRepository repository) // ServiceImpl зависит от IRepository

{

...

}

}

​

interface IRepository{}

class RepositoryImpl : IRepository

{

public RepositoryImpl(){} // может иметь свои зависимости, опустим для простоты

}

​

...

​

// конфигурация и использование контейнера

var dependencies = new DependenciesConfiguration();

dependencies.Register<IService, ServiceImpl>();

dependencies.Register<IRepository, RepositoryImpl>();

var provider = new DependencyProvider(dependencies);

​

// должен быть создан ServiceImpl (реализация IService), в конструктор которому передана

// RepositoryImpl (реализация IRepository)

var service1 = provider.Resolve<IService>();

...

​

Необходимо реализовать два варианта **времени жизни** зависимостей (задается при регистрации зависимости):

* **instance per dependency** — каждый новый запрос зависимости из контейнера приводит к созданию нового объекта;
* **singleton** — на все запросы зависимостей возвращается один экземпляр объекта (следует учитывать параллельные запросы в **многопоточной среде**).

Необходимо учитывать ситуацию наличия **нескольких реализаций** для **одной зависимости** и предусмотреть способ получения сразу всех реализаций. Например:

dependencies.Register<IService, ServiceImpl1>();

dependencies.Register<IService, ServiceImpl2>();

var provider = new DependencyProvider(dependencies);

IEnumerable<IService> services = provider.Resolve<IEnumerable<IService>>();

// ↑ должен вернуться IEnumerable с ServiceImpl1 и ServiceImpl2

Таким же образом IEnumerable<IService> должен создаваться, если он присутствует в **конструкторе другого класса**.

Зависимость может иметь **шаблонный тип**, в частности, тип, который влияет на конкретные типы ее зависимостей:

interface IService<TRepository> where TRepository : IRepository

{

...

}

​

class ServiceImpl<TRepository> : IService<TRepository>

where TRepository : IRepository

{

public ServiceImpl(TRepository repository)

{

...

}

...

}

В обычном варианте регистрация таких зависимостей визуально не отличается от не шаблонных:

dependencies.Register<IRepository, MySqlRepository>();

dependencies.Register<IService<IRepository>, ServiceImpl<IRepository>>();

Однако помимо этого должна быть доступна регистрация подобных зависимостей с помощью **open generics:**

dependencies.Register(typeof(IService<>), typeof(ServiceImpl<>));

Описанная зависимость является **параметризованной**: тип generic-параметра зависимости определяется только в момент запроса:

provider.Resolve<IService<IMySqlRepository>>()

// при создании ServiceImpl<TRepository> должен быть создана зависимость IMySqlRepository

// (объект класса, зарегистрированный в качестве реализации IMySqlRepository)

// и передана в конструктор

Достаточно реализовать параметризованные зависимости **первого порядка**, то есть когда open generic непосредственно является типом зависимости, а не параметром другого шаблона:

// open generics второго порядка

// поддержка таких случаев НЕ ТРЕБУЕТСЯ

// (код носит иллюстративный характер, такого синтаксиса нет, типы необходимо создавать через

// reflection вручную)

dependencies.Register(typeof(ICommand<IService<>>), typeof(MyCommand<ServiceImpl<>>));

Код лабораторной работы должен состоять из двух проектов:

* Dependency Injection контейнер;
* модульные тесты.

**Проверка работоспособности контейнера должна быть выполнена с помощью модульных тестов. Использовать вспомогательный проект с консольной программой запрещено.**

**Задание со звездочкой**

Добавить поддержку **именованных зависимостей**:

enum ServiceImplementations

{

First,

Second

}

​

dependencies.Register<IService, FirstImplementation>(ServiceImplementations.First);

dependencies.Register<IService, SecondImplementation>(ServiceImplementations.Second);

...

// получение напрямую

FirstImplementation first = container.Resolve<IService>(ServiceImplementations.First);

SecondImplementation second = container.Resolve<IService>(ServiceImplementations.Second);

...

// получение в конструкторе

public SomeAnotherService([DependencyKey(ServiceImplementations.Second)] IService service)

{

...

}

Именованные зависимости позволяют различать несколько реализаций одного интерфейса, когда это необходимо (в дополнение к возможности получения сразу всех реализаций).

Конкретный API получения именованных зависимостей на усмотрение автора, однако обязательна реализация явного получения через Resolve и в конструкторе.

### Дополнительная литература

* Dependency Injection in .NET, Марк Семанн