Проектирование и разработка модульных приложений

для платформы .NET

Цель

Уменьшение необязательной сложности разработки и сопровождения программного обеспечения

путем разбиения приложений на модули

Цель

Требования

Требования

- Слабая связанность: связи между модулями должны быть немногочисленными, явными, гибкими
- Ортогональность: средства для обеспечения модульности должны требовать только необходимый минимум изменений в приложении
- Повторное использование: один модуль может быть легко задействован в разных приложениях
- Компонуемость: легкость сборки своего приложения из набора модулей
- Рекурсивность: модуль сам должен допускать и поддерживать разбиение на модули

Цель

Требования

Микроскопы

PRISM

Суть

- Модуль - класс, реализующий маркерный интерфейс IModule, идентификация и зависимости настраиваются с помощью метаданных

Плюсы

- Порядок инициализации модулей автоматически определяется в соответствии с указанными зависимостями
- Официально рекомендованный, документированный и поддерживаемый Microsoft способ

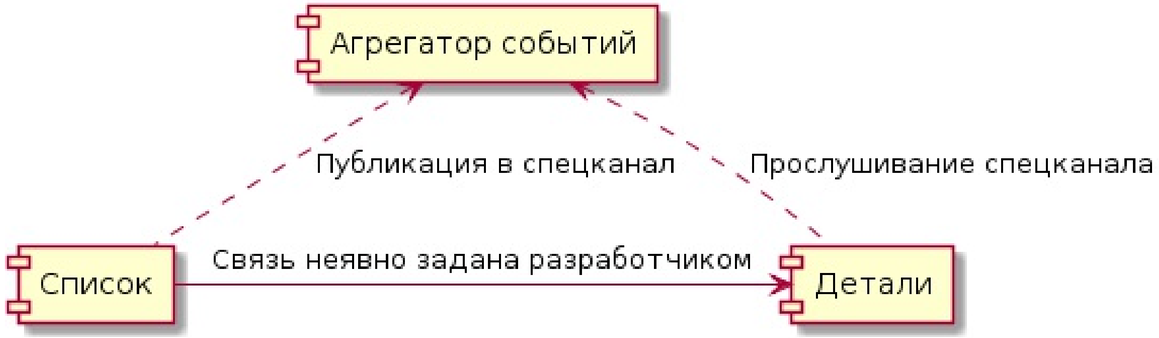
Минусы

- Организация финализации модулей целиком на плечах разработчика
- Зависимости модулей от ядра неявные
- Фактические зависимости модулей друг от друга могут не соответствовать заявленным метаданным и это никак не проверятся
- Поощряется использование эквивалентов глобальных разделяемых переменных (агрегатор событий, DI-контейнер)
- С рекурсивностью и компонуемостью по факту все очень плохо

Связь главный-детальный

- Решение в лоб

- Использование агрегатора событий



Суть

- Модуль - сборка, помеченная атрибутом [Addin], содержащая классы, помеченные специальным атрибутом [Extension], реализующие интерфейсы, помеченный атрибутом [TypeExtensionPoint]
- Модули обязательно должны привязываться к хосту (сборке, помеченной атрибутом [AddinRoot]) посредством атрибута [AddinDependency]

Плюсы

- Изначально реализовано как кросс-платформенная библиотека
- Именно на механизме Mono.Addins построена среда разработки SharpDevelop

Минусы

- Зависимости очень жесткие, включая зависимость от самой библиотеки
- Повторное использование модулей сильно затруднено

Суть

- Модуль - обычный класс, для сборки и конфигурации используется DI-контейнер

Плюсы:

- Автоматическое разрешение зависимостей
- Гибкое конфигурация точек сборки
- Настраиваемый контроль жизненного цикла модулей

Минусы:

- Очень сильная грануляция
- Нерекурсивность и недостаточная компонуемость - точка сборки с использованием контейнера не является таким же первоклассным модулем как класс
- Неявность межмодульных связей

Цель

Требования

Микроскопы

PRISM

Mono.Addins

DI-контейнеры

Микросервисы

Суть

- Выделение модулей в отдельные процессы

Плюсы

- Горизонтальная масштабируемость
- Зависимости только через контракты сервисов
- Приложение де-факто состоит из согласованно сконфигурированных сервисов

Минусы

- Невозможность использования эффективных внутрипроцессных коммуникаций
- Децентрализованная конфигурации приложения
- Сложность определения источника проблемы (ошибку выдает сервис A, а реальный сбой на стороне сервиса B)
- Зависимости между модулями неявные

Суть

- Выделение модулей в отдельные Nuget-пакеты

Плюсы

- Готовая инфраструктура с автоматическим контролем зависимостей
- Подключение нового пакета тривиально

Минусы

- Зависимости сильные и жесткие, даже если нужен один тип из пакета - будет зависимость на сам пакет и на все пакеты, от которых зависит он сам
- Исправленный код из nuget-пакета трудно отлаживать в итоговом приложении ввиду сложности доставки актуальных бинарников, если вы не разработчик этого пакета

Фатальный недостаток

ни одно из решений не ставит модульность во главу угла

Исходная конфигурация

- Два приложения (будет называть их альфа и браво), имеющие общий код для функциональности, не относящейся к бизнес-логике
- Средство разработки - Delphi 2007 (рефлексия очень слабая, обобщенных типов нет)
- Сервисы - COM-интерфейсы (IUnknown, GUIDs, AddRef, Release) без надстроек (OLE, ActiveX)

Требования от бизнеса

- Сделать возможным разработку третьего приложения (чарли) как дополнения к альфа, с доступом из чарли к любой необходимой функциональности альфы, включая бизнес-логику

Решение

- Выделение общего для всех ядра (сервера приложений)
- Использование размещенного в ядре Service Locator в качестве провайдера и регистратора сервисов для всех дополнительных модулей
- Хардкод для порядка загрузки

Результат

- Переход от монолитных приложений к модульным относительно дешевой ценой

Цель

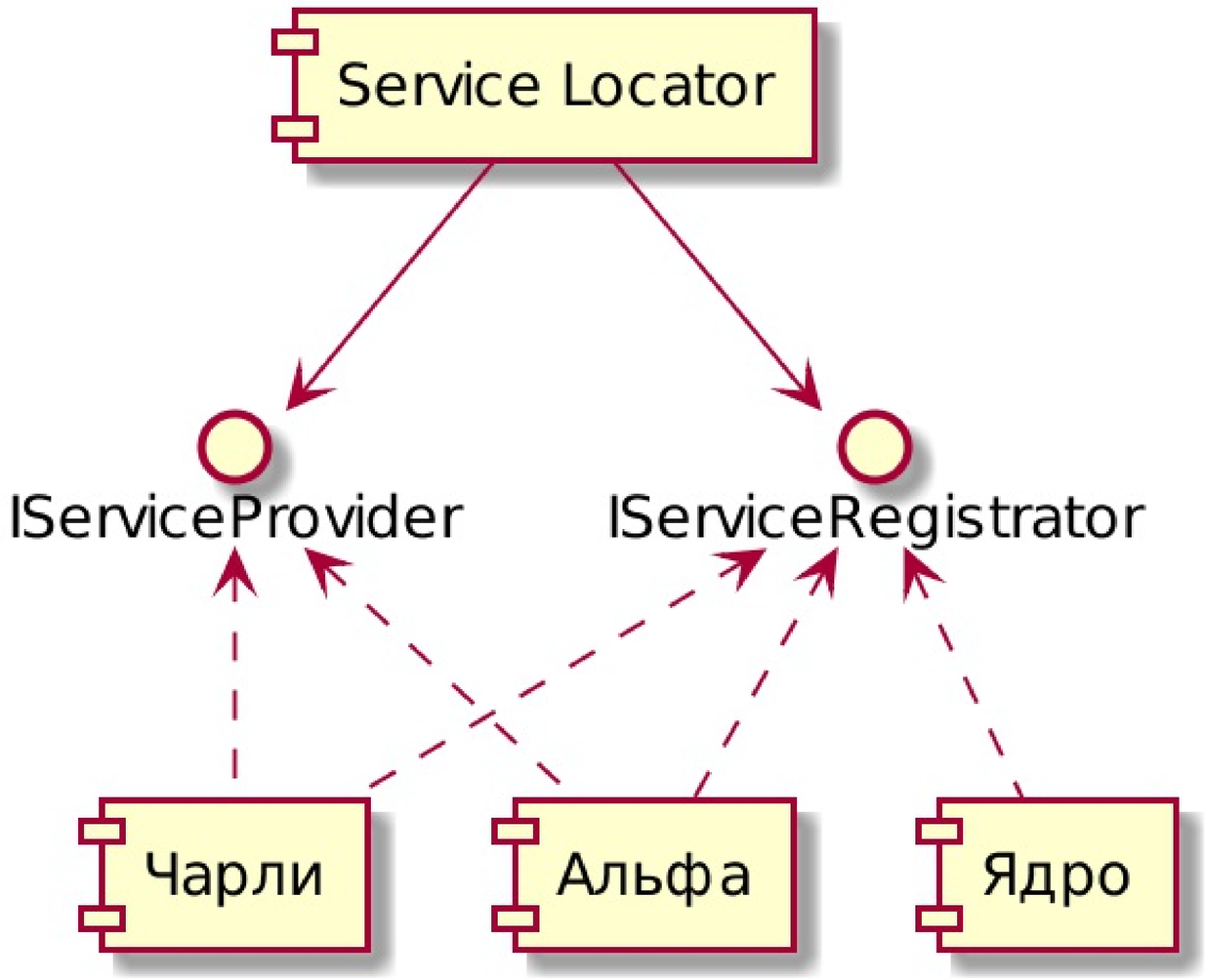
Требования

Микроскопы

Моя история

Первый блин

Использование общего локатора модулями



Новые требования

- Возможность поставки заказчикам различных конфигураций приложений как разного набора модулей. Варианты с поддержкой баз данных (MS SQL или Oracle), варианты с поддержкой резервирования и без, и т.п.
- Определять порядок инициализации модулей автоматически

Проблемы

- Необходимо иметь зависимости модулей друг от друга
- Нельзя иметь зависимости модулей друг от друга - в разных конфигурациях одни и те же функции исполняют разные модули

Решение

- Модули выставляют свои зависимости и реализуемые сервисы как интерфейсы в локальном для каждого модуля Service Locator
- Ядро способно, имея список модулей, автоматически привязать зависимости к реализациям, построить граф зависимостей, провести его топологическую сортировку, после чего инициализировать (и финализировать!) модули в правильном порядке

Результат

- Успешное разрешение контроля зависимостей модулей

Цель

Требования

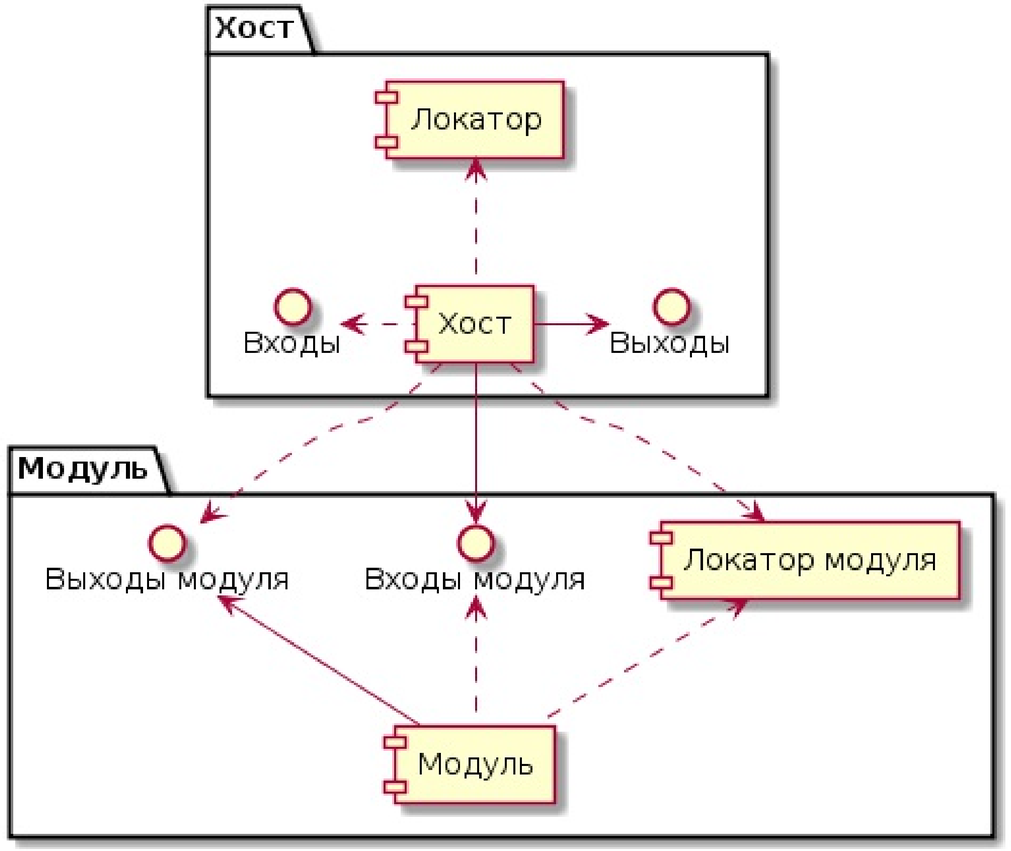
Микроскопы

Моя история

Первый блин

Зависимости

Взаимодействие модуля с хостом



Проблема

- Модули поддержки MS SQL и Oracle должны быть инициализированы до первого реального обращения к базе данных
- От модулей поддержки конкретных СУБД ни один бизнес-модуль не зависит
- Модуль, реализующий сервисы работы с базой - также о поддержке конкретных СУБД ничего не знает

Решение

- Реализацию поддержки модульности как фабрики ядер - в результате кто угодно может стать ядром для своего списка модулей
- Построив граф зависимостей для своих подмодулей, модуль может выставить их оставшиеся нереализованными входные зависимости как собственные

Результат

- Реализация всех приложений компании как набора из десятков модулей, из которых только единицы были уникальными для конкретного приложения
- Не потребовалось никаких изменений самого механизма модульности при сколь угодно сложных требований к приложениям

Цель

Требования

Микроскопы

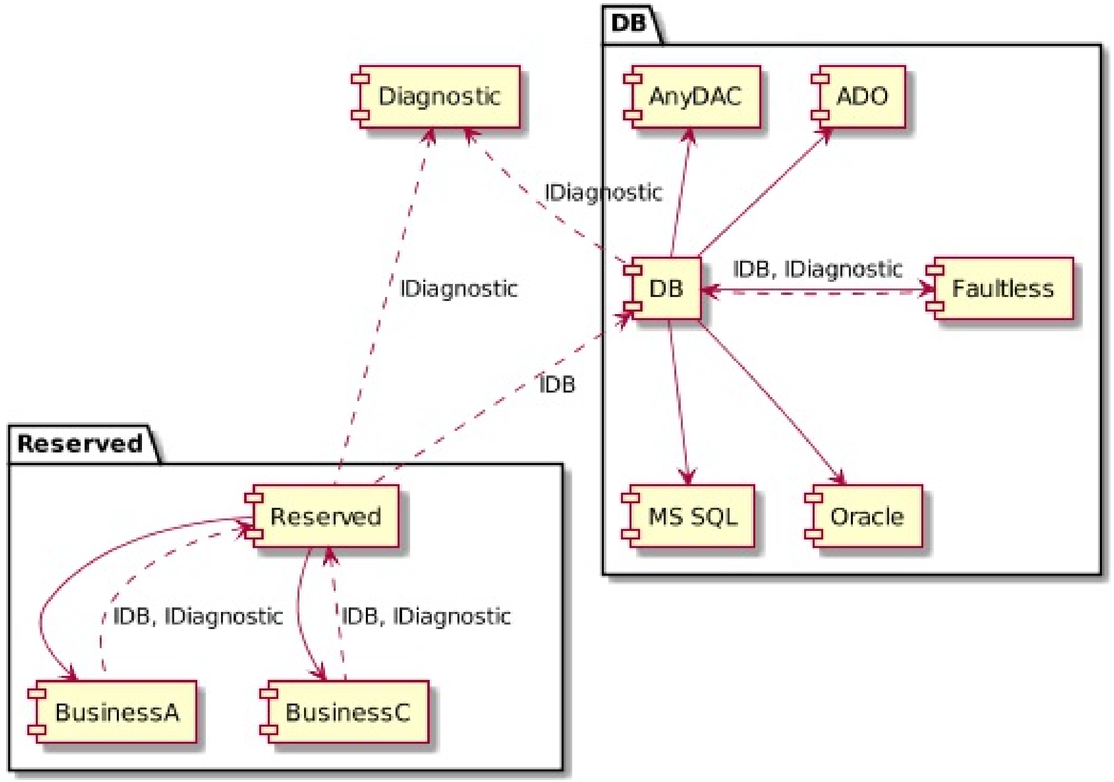
Моя история

Первый блин

Зависимости

Рекурсивность

Взаимодействие модулей с подмодулями



Современное решение (NetStandard 1.1)

- Модуль - экземпляр класса (не класс, а объект!), реализующий специальный интерфейс IModule

```
// Активация
// Получает поставщик входных зависимостей модуля
// Возвращает поставщик выходных как успешный результат
// Бросает исключение при неудаче
// Для деактивации достаточно вызвать Dispose у результата
Usable<IDependencyProvider> Activate(
   IDependencyProvider dependencies);

// Неизменяемая информация о модуле - описатель модуля
IModuleDescriptor Descriptor { get; }
```

- Описатель модуля (интерфейс IModuleDescriptor)

```
// Человекочитаемое имя модуля
string Name { get; }

// Машинно-читаемый уникальный идентификатор модуля
Guid Id { get; }

// Входные зависимости модуля
// Сервисы, реализация которых требуется модулю для работы
IImmutableSet<Type> Input { get; }

// Выходные зависимости модуля
// Сервисы, которые модуль реализует сам
IImmutableSet<Type> Output { get; }
```

- Входные и выходные зависимости на примере класса

```
public sealed class Module :
    IServiceA, IServiceB // Выходные зависимости

    public Module(
        IServiceC c, IServiceD d) // Входные зависимости
```

Цель

Требования

Микроскопы

Моя история

Молоток

Модуль

Общее между модулем и обычным классом

```
// IModuleDescriptor.Name, IModuleDescriptor.Id
public sealed class Module :

    // IModuleDescriptor.Output
    // IModule.Activate.returned.IDependencyProvider
    IServiceA, IServiceB,

    // IModule.Activate.returned.Usable<>
    IDisposable
{
    // IModule.Activate
    public Module(

        // IModuleDescriptor.Input
        // IModule.Activate.dependencies
        IServiceC c, IServiceD d
```

Различия между модулем и обычным классом

- Модуль - это объект, а не класс
- Имея один класс можно создать несколько модулей
- У класса уникально полное имя, у модуля уникален Id
- Зависимости класса определяются при компиляции, зависимости модуля - во время его создания при выполнении
- Конструктор каждый раз создает новый объект, метод Activate меняет состояние самого модуля
- Конструктор можно вызывать многократно, метод Activate бросает исключение, если модуль уже активирован
- Для создания класса во время исполнения нужна кодогенерация, для создания модуля достаточно дескриптора с уникальным Id и метода активации

Ключевое звено для работы модулей - поставщик зависимостей (интерфейс IDependencyProvider)

```
// Разрешение зависимости - возврат реализации типа
// Бросает исключение,
// если заказанной реализации нет в списке зависимостей
// Для освобождения от зависимости - вызов Dispose()
Usable<object> Resolve(Type type);

// Зависимости, для которых предоставляются реализации
// Не модули, а интерфейсы!
IImmutableSet<Type> Dependencies { get; }
```

Сравнение с IServiceProvider

- В обоих случаях список поддерживаемых типов формируется в период исполнения
- Поставщик зависимостей имеет типизацию периода выполнения (список поддерживаемых типов доступен и неизменен на время жизни поставщика)
- Использованные зависимости можно возвращать (вызвав Usable.Dispose())

Граф зависимостей модулей - интерфейс IModuleGraph

```
// Внутренние зависимости в графе модулей
// зависимый модуль -> [реализующий модуль -> типы]
IImmutableDictionary<
    IModuleDescriptor, ILookup<IModuleDescriptor, Type>>
    InnerDependencies { get; }

// Внешние входные зависимости
// тип -> зависимые модули
ILookup<Type, IModuleDescriptor> Input { get; }

// Внешние выходные зависимости
// тип -> реализующие модули
ILookup<Type, IModuleDescriptor> Output { get; }

// Порядок активации модулей (если определен)
IImmutableList<IModuleDescriptor> Order { get; }

// Цикл зависимостей (если присутствует)
IImmutableList<KeyValuePair<IModuleDescriptor, IEnumerable
    Cycle { get; }
```

- Содержит все межмодульные зависимости в явном виде
- Не меняет состояние модулей - использует только дескрипторы
- Взаимно однозначно отображается на диаграмму компонентов UML

Построение графа зависимостей модулей - метод расширения ToModuleGraph

```
public static IModuleGraph ToModuleGraph(
    // Нужны только дескрипторы!
    this IEnumerable<IModuleDescriptor> modules,
    // Выбор внутренней связи между модулями
    Func<
        // Зависимый модуль, интерфейс, реализующие модули
        IModuleDescriptor, Type, IEnumerable<IModuleDescri
        // Результат - выбранный реализующий модуль (или n
        IModuleDescriptor> tryChoiceImplementation)
```

- Это чистая функция, возвращающая неизменяемый результат
- Межмодульные связи создаются автоматически
- Выбор конкретного варианта связи для конкретного модуля управляется с помощью делегата tryChoiceImplementation. Простейший вариант выбора:

```
(module, dependencies, implementations) =>
implementations.First()
```

- Неразрешенные входные зависимости модулей формируют входные зависимости графа
- Выходные зависимости графа составляются из всех выходных зависимостей модулей

Цель

Требования

Микроскопы

Моя история

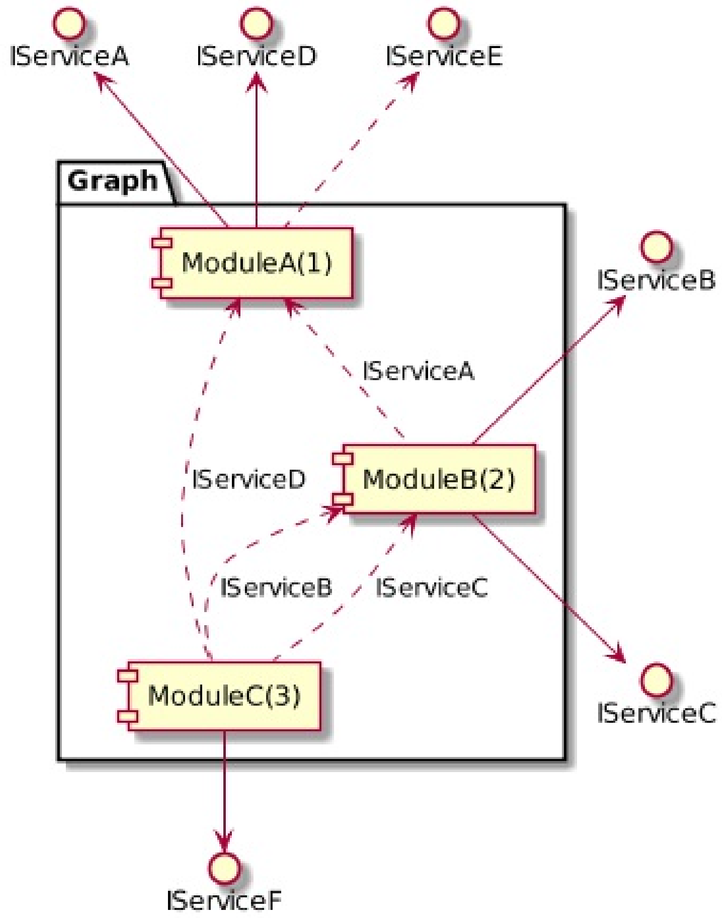
Молоток

Модуль

Зависимости

Граф

Изображение графа зависимостей модулей как компонентной диаграммы UML



- Интерфейс IServiceE является входной зависимостью для графа модулей
- Остальные интерфейсы - выходные зависимости
- Число в скобках - порядковый номер для активации

Создание модуля верхнего уровня на основе графа зависимостей - метод расширения ToModule

```
public static IModule ToModule(this IModuleGraph graph,
    Func<
        // Зависимость (интерфейс), реализующие модули
        Type, IEnumerable<IModuleDescriptor>,
        // Выбранный реализующий модуль или null
        IModuleDescriptor> tryChoiceOutput,
    // Имя надмодуля
    string name,
    // Уникальный идентификатор
    Guid id,
    // Словарь описатель - модуль
    IImmutableDictionary<IModuleDescriptor, IModule> modul
```

- Модуль верхнего уровня выставляет все входные зависимости графа как свои собственные
- Выходные зависимости можно выбрать и отфильтровать как посчитает нужным разработчик
- При активации надмодуля все модули графа будут активированы в правильном порядке

- Контроллер - активация только необходимых в данный момент модулей по запросу с помощью интерфейса IModuleController

```
// Описатели управляемых контроллером модулей
IEnumerable<IModuleDescriptor> Modules { get; }

// Получение поставщика зависимостей модуля по описателю
// Автоматическая активация и деактивация модулей,
// включая те, от которых зависит указанный
Usable<IDependencyProvider> GetProvider(
    IModuleDescriptor descriptor);

// Проверка состояния модуля
bool IsActive(
    IModuleDescriptor descriptor);

// Событие об изменении состояния модуля
IObservable<KeyValuePair<IModuleDescriptor, bool>>
    ActiveChanged { get; }
```

```
public static IModuleController ToModuleController(
    // Граф зависимостей модулей
    this IModuleGraph graph,
    // Модули с доступом по описателю
    IImmutableDictionary<IModuleDescriptor, IModule> modul
    // Провайдер входных зависимостей
    IDependencyProvider input)
```

Интеграция модулей с Autofac

- Реализация модуля как конфигурация точки сборки Autofac

```
// Создание модуля из контейнера Autofac
public static IModule ToAutofacModule(
// Дескриптор модуля -
// Выходные зависимости обязаны регистрироваться в кон
this IModuleDescriptor descriptor,
// Привычная настройка контейнера
Action<ContainerBuilder> registrator)
```

- Регистрация модуля как компонента в Autofac

```
// Регистрация выходных зависимостей модуля как сервис
public static void RegisterFluentHeliumModule(
// Конфигурируемый контейнер
this ContainerBuilder builder,
// Модуль для регистрации
IModule module,
// Список сервисов для регистрации
// По умолчанию регистрируются все выходные зависимост
IEnumerable<Type> types)
```

Результаты

- Полный контроль зависимостей внутренних (дочерних) контейнеров от внешних
- Готовое решение для ASP.NET Core
- Образец для легкой интеграции с другими DI-контейнерами

Цель

Требования

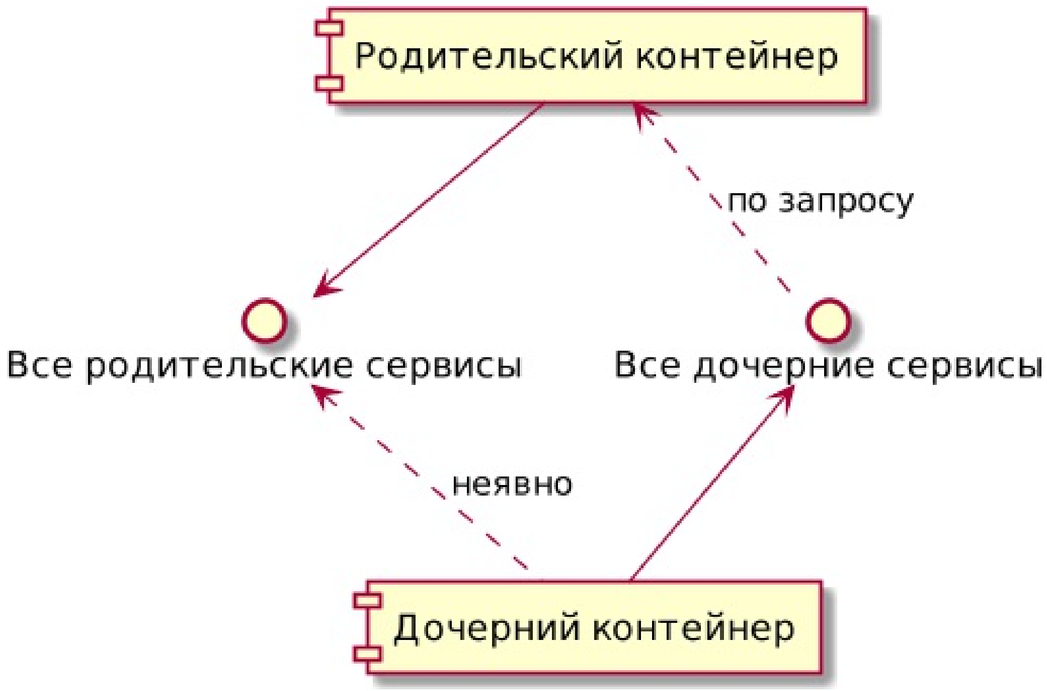
Микроскопы

Моя история

Молоток

Интеграция

Традиционный вариант использования дочерних DI-контейнеров



- Дочерний контейнер наследует все сервисы родительского
- Дочерний контейнер предоставляет все свои сервисы родительскому
- Дочерний контейнер должен быть реализован той же библиотекой, что и родительский

Цель

Требования

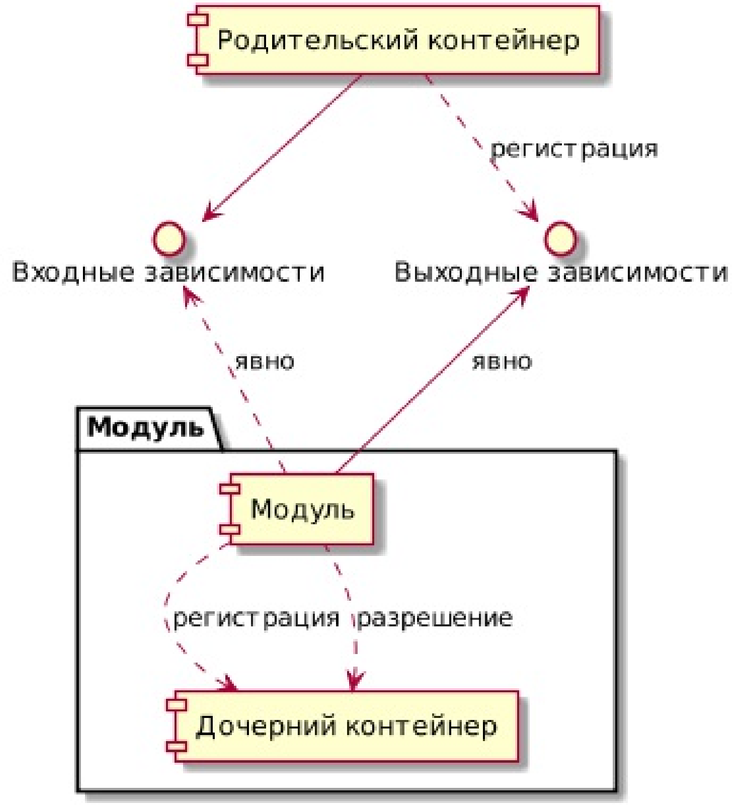
Микроскопы

Моя история

Молоток

Интеграция

Подключение дочерних DI-контейнеров с использованием модуля



- Дочерний контейнер использует явно перечисленные сервисы родительского (входные зависимости модуля)
- В родительском контейнере регистрируется явно указанное подмножество сервисов дочернего (выходные зависимости модуля)
- Реализация дочернего контейнера может не иметь ничего общего с родительским

Цель

Требования

Микроскопы

Моя история

Молоток

Интеграция

Планы

- Поддержка необязательных зависимостей (Option)
- Поддержка множественных зависимостей (IEnumerable)
- Добавление примера модульного приложения
- Использование в боевом проекте
- Визуализация в WPF

Плюсы

- Модуль сам по себе не зависит от других модулей, его легко разрабатывать, тестировать и поддерживать независимо
- Все зависимости модуля - явные по построению. Даже неявные де-факто зависимости (платформа, требование сборки по сильному имени для активации и т.п.) можно выразить с помощью маркерных интерфейсов
- В модуль можно обернуть все что угодно - класс, обычную сборку, сборку с ленивой загрузкой, сборку с выгрузкой при активации, сконфигурированный DI-контейнер, внешний сервис и т.п.
- В точке сборки связи между модулями создаются автоматически, а порядок активации определяется топологической сортировкой
- Граф зависимостей модулей взаимно однозначно отображается на диаграмму компонентов UML (конвертация графа модулей в PlantUML уже есть в библиотеке)
- Ненавязчивость - любая часть библиотеки допускает легкую замену самописным аналогом

Минусы

- Обертка в виде модуля для кода, который заранее не спроектирован под модульность, может оказаться весьма дорогой и нетривиальной
- В настоящий момент это велосипед без поддержки от вендоров и сообщества

Источники вдохновения

- Однозначно рекомендуемая книга о внедрении зависимостей в .NET (ссылки на оригинал и неофициальный перевод)
- Отрисовка диаграммы PlantUML онлайн
- Актуальный репозитарий Mono.Addins на гитхабе
- "Модули" от Autofac
- "Поддержка" Dependency Injection в APS.NET Core
- Интеграция Autofac с ASP.NET Core
- Моя микробиблиотека на Bitbucket с открытым исходным кодом (лицензия MIT)
- Подробности про Usable (на тот момент еще IDisposable)
- Про написание юнит-тестов для моей библиотеки

Благодарности

- Организаторам в лице Юлии Цисык за приглашение выступить с докладом
- Никите Цуканову и коллегам из сообщества .NET разработчиков за ценные замечания и рекомендации по теме доклада
- Илье Ефимову за совместную работу на Дойчебанк, откуда было почерпнуто множество идей
- Коллегам из самарской компании СМС-ИТ за первый успешный опыт разработки модульных приложений

Вопросы?

Маурин Кирилл

leo.bonart@gmail.com

bitbucket.org/KirillMaurin

Сделано с помощью remark