Санкт-Петербургский государственный университет

Фундаментальная информатика и информационные технологии

Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Николаев Сергей Юрьевич

Аспектно-ориентированная реализация принципа инверсии зависимости при разработке программного обеспечения

Магистерская диссертация

Научный руководитель:

к. ф.-м. н., доц. Григорьев Д. А.

Рецензент:

Разработчик, ООО “Яндекс” Макаров Сергей Алексеевич

Санкт-Петербург

2017

SAINT-PETERSBURG STATE UNIVERSITY

Fundamental Computer Science and Information Technologies

Mathematical and Software Support for Computers, Computer Systems and Networks

Sergei Nikolaev

Aspect-oriented implementation of dependence inversion principle in developing software.

Master's Thesis

Scientific supervisor:

cand. of Sc., associate professor Dmitry Grigoryev

Reviewer:

Developer, OOO (Limited Liability Company) “Yandex” Makarov Sergei Alexeyevich

Saint-Petersburg

2017

Оглавление

[1.1. Введение 4](#_Toc483489862)

[1.2. Задача 4](#_Toc483489863)

[1.3. Актуальность 5](#_Toc483489864)

[1.4. Обзор данной работы 6](#_Toc483489865)

[2.1. Внедрение зависимостей(Dependency Injection) 7](#_Toc483489866)

[2.2. Пример «Hello DI!!» 8](#_Toc483489867)

[2.3. Методы внедрения зависимостей 9](#_Toc483489868)

[2.3.1. Внедрение через конструктор (Constructor Injection) 9](#_Toc483489869)

[2.3.2. Внедрение через метод (Method Injection) 10](#_Toc483489870)

[2.3.3. Внедрение через свойство(Property Injection) 10](#_Toc483489871)

[2.4.1 IOC-контейнеры 11](#_Toc483489872)

[2.4.2. Управление жизненным циклом объектов и механизм перехвата 12](#_Toc483489873)

[2.5 Обзор существующих IOC-контейнеров 14](#_Toc483489874)

[2.5.1 Unity 15](#_Toc483489875)

[2.5.2 Spring.NET 16](#_Toc483489876)

[2.5.3 Краткий обзор остальных продуктов 16](#_Toc483489877)

[3.1. ASPECT.NET 17](#_Toc483489878)

[4.1. MS Roslyn 19](#_Toc483489879)

[4.1.1. SyntaxTree 19](#_Toc483489880)

[4.2. Элементы SyntaxTree 21](#_Toc483489881)

[4.2.1 SyntaxNode 21](#_Toc483489882)

[4.2.2 SyntaxToken 21](#_Toc483489883)

[4.2.3 SyntaxTrivia 22](#_Toc483489884)

[4.3 Модификация исходного кода 22](#_Toc483489885)

[4.4 Семантика 23](#_Toc483489886)

[4.4.1 Compilation 23](#_Toc483489887)

[4.4.2 Symbol 23](#_Toc483489888)

[4.4. Semantic model 24](#_Toc483489889)

[4.5. Workspaces 25](#_Toc483489890)

[5.1. Реализация. Схема проекта. 25](#_Toc483489891)

[5.1.1. Вынесение IOC-контейнера в аспекты. 28](#_Toc483489892)

[5.2 Анализ и рефакторинг посредством MS Roslyn 31](#_Toc483489893)

[6. Заключение 37](#_Toc483489894)

[6.1. Результаты данной работы 37](#_Toc483489895)

[6.2. Направление дальнейших исследований 37](#_Toc483489896)

[7. Список литературы 38](#_Toc483489897)

[8. Приложение 40](#_Toc483489898)

## Введение

Паттерн DI (*dependency injection*) и IoC-контейнеры[6] давно стали неотъемлемой частью любых крупных проектов. Благодаря им можно построить так называемый слабосвязанный код. Inversion of Control – это паттерн, поручающий компоненте управление каким-нибудь объектом. А Dependency Injection позволяет автоматически получить из контейнера нужные нам зависимости при инициализации. Ими активно пользуются многие программисты. Представителей данных библиотек на данный момент очень много, в проекте рассмотрены IOC-контейнеры Ninject[10], Unity[9], Autofac[12], MEF[11] и другие. Все они постоянно развиваются, пытаясь улучшить существующую модель.

### Задача

Задача состоит в реализации принципа инверсии зависимости[6] с использованием ASPECT.NET[5] и вынесении IOC-контейнеров в аспекты. Помимо методологии ручного вынесения контейнеров требуется автоматизация данной теории посредством технологии MS Roslyn[18].

### Актуальность

Использование слабого связывания паттерна DI даёт ряд преимуществ, таких как: позднее связывание, расширяемость, параллельная разработка, удобство сопровождения, тестируемость[3], которые ценятся всегда.

**Позднее связывание** делает программу более гибкой и добавляет возможность заменять один сервис другим[6]. Для многих это является самым преобладающим преимуществом, но не стоит взвешивать все достоинства и недостатки, имея ввиду только его.

Возможность **расширить**, модифицировать программу также является залогом успешного программного обеспечения. Слабое связывание позволяет эффективно перестраивать приложение и вносить в него новые возможности[3].

Отсутствие сильной связанности позволяет использовать концепцию разделения, делающая возможным **разработку параллельными командами**[6], которая необходима при достижении проекта определённого размера.

Программу, написанную с использование паттерна DI очень легко **сопровождать**, потому что здесь используется **принцип единственной ответственности** (*single responsibility principle*)[6], утверждающий, что каждый класс должен обладать только одной ответственностью. Теперь, зная кто за что отвечает, нам легко будет понять, где вести изменения, кого наказать при неисправности и так далее.

С **тестированием** все очевиднее. Слабое связывание по определению разрешает модульное тестирование.

И не нужно забывать про **ASPECT.NET**[5], который очень хорошо подходит для решения нашей задачи, так как IOC-контейнеры, являясь сквозной функциональностью, могут быть там очень удобно реализованы, либо вынесены.

### Обзор данной работы

В следующих разделах рассматривается паттерн DI, в том числе программные продукты, реализующие его, и методы, благодаря которым зависимости внедряются в бизнес-логику проекта. Также будет говориться об IOC-контейнерах – библиотеках, отвечающих за инициализацию паттерна, и рассмотрены и сравнены различные существующие контейнеры различных компаний. В главе 3 рассматривается ASPECT.NET разработка кафедры информатики СПбГУ, с помощью которой и проводится «бесшовная» интеграция IOC и бизнес-логики.

В следующих главах рассматривается продукт под названием MS Roslyn, благодаря которому мы можем получить доступ к синтаксическому дереву и разобраться с семантикой любой бизнес-логики, включающей в себя инициализацию паттерна DI. Благодаря этому появляется возможность анализировать этот код, генерировать новый и позволить ASPECT.NET используя эти данные автоматически распознавать искомую информацию об IOC-контейнерах и выносить её в аспекты.

## Внедрение зависимостей(Dependency Injection)

**Механизм внедрения зависимостей** (*Dependency Injection*) – шаблон проектирования для создания слабосвязанного кода. Для понимания этого определения существует хороший пример из книги «Внедрение зависимостей в .NET» [6]: вопрос «Как объяснить механизм внедрения зависимостей 5-летнему ребенку?». Вот аналогия Джона Манша:

*Когда ты идешь к холодильнику и что-то самостоятельно достаешь из него, ты можешь стать причиной возникновения проблем. Ты можешь оставить дверь открытой, ты можешь взять то, что мама или папа не хотели бы, чтобы ты брал. Ты даже можешь искать то, чего у нас и не было, или то, что уже закончилось.*

*Все, что тебе следует сделать, – это сформулировать, что тебе нужно, "Мне нужно то-то, чтобы выпить во время ланча", и после этого мы будем следить за тем, чтобы у тебя было это, когда ты садишься кушать.*

То есть взаимодействующие классы (5-летние дети) полагаются на инфраструктуру (родителей), обеспечивающую эти классы искомыми сервисами.

**Слабосвязанный код**. **Степень связанности** (*coupling*) — это мера, определяющая насколько жестко один элемент связан с другими элементами, либо каким количеством данных о других элементах он обладает. Элемент с низкой степенью связанности (или слабым связыванием) зависит от не очень большого числа других элементов [3]. Задача – избавиться от зависимостей. Хорошо описывает эту задачу ключевое правило "Банды четырех" (Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидс) из их книги "Паттерны проектирования" ("Design Patterns")[1],

*Программируй, основываясь на интерфейсе, а не на классах.*

Слабая степень связанности позволяет расширять код, а следовательно делает его поддерживаемым.

### Пример «Hello DI!!»

Простейший пример из книги «Внедрение зависимостей .NET» [6].

Метод *main* и его взаимодействующие классы:

|  |  |
| --- | --- |
| 123456 | **private** **static** **void** **Main**(){ IMessageWriter writer = **new** ConsoleMessageWriter(); **var** salutation = **new** Salutation(writer); salutation.Exclaim();} |

ConsoleMessageWriter() – ввод данных, и экземпляр класса Salutation записывающий в него свой вывод.

Графическое представление ситуации:

ы

Использует

Использует

Создает

Salutation

Main()

IMessageWriter

ConsoleMessageWriter

Класс Salutation:

|  |  |
| --- | --- |
|  1 2 3 4 5 6 7 8 910111213141516 | **public** **class** **Salutation**{ **private** **readonly** IMessageWriter writer; **public** **Salutation**(IMessageWriter writer) { **if** (writer == **null**) { **throw** **new** **ArgumentNullException**("writer"); } **this**.writer = writer; } **public** **void** **Exclaim**() { **this**.writer.Write("Hello DI!"); }} |

В конструктор вместо конкретного класса ставится его интерфейс, тем самым получается слабосвязанный код. Этот пример характеризует метод **внедрения через конструктор** (*Constructor Injection*).

Интерфейс:

|  |  |
| --- | --- |
| 1234 | **public** **interface** IMessageWriter{ **void** **Write**(**string** message);} |

Класс, его реализующий:

|  |  |
| --- | --- |
| 1234567 | **public** **class** **ConsoleMessageWriter** : IMessageWriter{ **public** **void** **Write**(**string** message) { Console.WriteLine(message); }} |

Добавление интерфейса для работы со слабосвязанным кодом вызывают издержки в размере 1,1%, которые с ростом кода становятся незаметными, а на выходе получаем грамотно написанный хорошо расширяемый код.

### Методы внедрения зависимостей

### Внедрение через конструктор (Constructor Injection)

**Внедрении через конструктор**[8] из предыдущего примера. Схема:

|  |  |
| --- | --- |
| 12345678 | **public** **class** **ClassName** { **private** **readonly** AnotherServiceInterfaсe \_service; **public** **ClassName**(AnotherServiceInterfaсe service) { \_service = service; }} |

### Внедрение через метод (Method Injection)

Является обобщением предыдущего метода [8]**.** Схема:

|  |  |
| --- | --- |
| 12345678 | **public** **class** **ClassName**{ **public** returnType **MethodName**(AnotherServiceInterface service)  { service.doWork(); }} |

### Внедрение через свойство(Property Injection)

Используется для сервисов с реализацией по умолчанию. Приведем пример из личного блога Сергея Теплякова [17]. Реализация логгера:

|  |  |
| --- | --- |
|  1 2 3 4 5 6 7 8 9101112131415161718192021 | **interface** ILogger{ **void** **Write**(**string** message);} **class** **DefaultLogger** : ILogger{ } **class** **Service**{ **private** **readonly** ILogger \_logger =  **new** **DefaultLogger**(); **public** **Service**() { }  **public** **Service**(ILogger logger) { \_logger = logger; }} |

### IOC-контейнеры

**IOC-контейнер** – это библиотека, которая обеспечивает функциональность механизма внедрения зависимостей. Она может автоматизировать множество задач, включая компоновку объектов и управление их жизненным циклом. Проще говоря, контейнер знает обо всех интерфейсах и их реализациях в системе и умеет их сопоставлять. Перед началом работы с контейнером регистрируются известные типы и их сопоставления.

Ключевым методом IOC-контейнеров является метод Resolve() (Unity, Ninject), или GetInstance() (Castle Windsor).

|  |  |
| --- | --- |
| 1 | **object** **Resolve**(Type service); |

Подставляя в этот метод аргумент, класс основного сервиса, мы получим экземпляр требуемого типа, который уже будет заполнен нашими зависимостями. Но как он поймет, как друг от друга зависят наши сервисы? Для начала мы должны сами ему об этом сообщить, сконфигурировать его. Существует 3 способа конфигурации: с помощью конфигурационного файла (XML), динамически (прямо в коде) и с помощью атрибутов. Рассмотрим конфигурацию прямо в коде. В этом нам поможет метод **RegisterType**<IService, Service>(), сопоставляющий наш интерфейс и конкретную реализацию. Теперь обладая таким знанием мы можем переписать наш пример «Hello DI!!!», используя контейнер Unity:

|  |  |
| --- | --- |
| 123456789 | **private** **static** **void** **Main**(){ **var** unityConteiner = **new** UnityContainer(); unityConteiner.RegisterType<IMessageWriter, ConsoleMessageWriter()>(); **var** salutation = unityContainer .Resolve<Salutation>(); salutation.Exclaim();} |

### 2.4.2. Управление жизненным циклом объектов и механизм перехвата

Используя IOC-контейнер, мы позволяем ему самому создавать зависимости, тем самым теряется возможность контролировать их **жизненный цикл.** Однако в дополнительные возможности почти всех современных контейнеров входит возможность вмешиваться в создание объектов и их контроль. Для этого используются паттерны стилей существования, определяющие новое поведение создающихся экземпляров[6]:

| **Название** | **Описание** |
| --- | --- |
| Singleton | Один экземпляр постоянно повторно используется |
| Transient | Всегда используются новые экземпляры |
| Per Graph | Один экземпляр повторно используется в пределах каждой диаграммы объектов |
| Web Request Context | В большинстве случаев для одного веб-запроса используется один экземпляр каждого типа |
| Pooled | Используются экземпляры из пула готовых объектов |
| Lazy | Зависимость, требующая больших затрат, создается и используется в замедленном темпе |
| Future | Зависимость станет доступна в будущем |

Например, чтобы в IOC-контейнере Autofac вместо стандартого поведения при создании экземпляра нашего сервиса использовать стиль существования Singleton (соответствует паттерну Singleton), нужно во время конфигурации просто изменить стандартный код, задающий соответствие между интерфейсом и классом:

|  |
| --- |
| builder.RegisterType<ContractMapper>() .As<IContractMapper>();  |

на следующий код:

|  |  |
| --- | --- |
|  | builder.RegisterType<ContractMapper>() .As<IContractMapper>().**InstancePerDependency**();  |

Что касается **механизма перехвата**, то его концепция довольно проста: нам хотелось бы уметь перехватывать сигнал между потребителем и сервисом, и исполнять некоторый код до или после вызова данного сервиса. Для реализации этой концепции наилучшим образом подходит паттерн **Decorator**[1], так как по принципу единственной ответственности наш сервис не должен реализовывать новую функциональность[3], а Decorator не вносит в него изменения, а лишь динамически добавляет новый код. Тем самым мы получаем возможность выносить сквозную функциональность с помощью IOC-контейнера. Значит с помощью контейнера можно аккуратно реализовать обработку ошибок, логирование и другие подобные сервисы. Схема перехвата выглядит так:

 

Посредник получает обычный сигнал и передает его в действующую реализацию, при этом действуя на сигнал так, чтобы он выполнял то, что ему требуется делать.

### Обзор существующих IOC-контейнеров

Здесь приводится таблица, описывающая самые популярные на данный момент IOC-контейнеры[6].

| **Название** | **Организация** | **Комментарии** |
| --- | --- | --- |
| Castle Windsor | Open Source | Состоявшийся и широко используемый |
| StructureMap | Open Source | Состоявшийся и широко используемый |
| Spring.NET | SpringSource | Состоявшийся и широко используемый порт Java Spring DI-контейнера |
| Autofac | Open Source | Наиболее современный DI-контейнер[12], созданный на основании характеристик языка C# 3.0 |
| Unity | Microsoft patterns&practices | Первая роль компании Microsoft в пространстве DI, Но не поддерживается ею.[9] |
| Managed Extensibility Framework (MEF) | Microsoft | Поставляется вместе с .NET 4, но в действительности является не DI-контейнером, а фреймворком расширяемости (как видно из его названия Managed Extensibility Framework)[11] |
| Ninject | Open Source | При наличии полного функционала, его API легок в применении[10] |

Для того, чтобы выбрать правильный IOC-контейнер для той или иной ситуации, не достаточно простой технической оценки. Нужно оценить, доверяете ли вы организации, разработавшей его, подходит ли вам лицензионная модель и так далее. Далее будут рассмотрены некоторые реализации IOC-контейнеров.

### Unity

Unity – контейнер, разработанный группой Patterns&Practices компании Microsoft[9]. Будучи «полуофициальным детищем Microsoft», является очень популярным продуктом. Контейнер был создан в мае 2008 года и стал поводом многих молодых программистов начать знакомство с темой внедрения зависимостей. Одним из самых важных плюсов продукта является наличие официальной документации. Обладает постоянным API, но страдает из-за отсутствия гибкости: помимо необязательных имен и менеджеров управления жизненным циклом, вся конфигурация выполняется с помощью классов, унаследованных от абстрактного класса *InjectionConstructor*. Всё это может привести к усложнениям проверки. Помимо императивного API обладает XML схемой, что позволяет использовать конфигурацию как в коде, так и в XML. Еще одним плюсом является наличие механизма перехвата. Из минусов: отсутствует поддержка механизма автоматической регистрации, позволяющей регистрировать тот или иной сервис без предварительной конфигурации, и слабый механизм управления жизненным циклом.

В общем Unity – прекрасный IOC-контейнер, у него есть некоторые минусы, но он предоставляет полный набор возможностей, а отсутствующую автоматическую регистрацию при желании можно и дописать самому.

### Spring.NET

Контейнеры Autofac, Ninject очень похожи на предыдущий Unity, их можно отнести к контейнерам второго поколения, поэтому рассмотрим непохожий на остальные Spring.NET[14]. Это единственный продукт реализованный как порт из Java. Очевидно, что он в большей степени полагается на XML-конфигурацию. На сегодняшний день использование этого контейнера несет за собой большое количество XML-кода. Плюсами является наличие механизма перехвата, полноценный пакет документации и возможность покупки коммерческого сопровождения. Минусы: слабый механизм сопровождения жизненным циклом, ограниченные возможности автоматической регистрации и чрезмерная завязанность на XML.

### Краткий обзор остальных продуктов

Здесь представлена таблица с описанием других популярных контейнеров[6]:

| **DI-контейнер** | **Достоинства** | **Недостатки** |
| --- | --- | --- |
| Castle Windsor | ЗавершенностьПонимает DecoratorДоступно коммерческое сопровождение | Местами извилистое API |
| StructureMap | Просто работает в большинстве случаев | Не поддерживает механизм перехвата |
| Autofac | Простое для изучения APIДоступно коммерческое сопровождение | Не поддерживает механизм перехватаЧастично поддерживает пользовательские жизненные циклыИспользует лямбда-выражения для конфигурации |
| MEF | Доступен в .NET 4/Silverlight 4 BCLКоммерческое сопровождение | Не настоящий DI-контейнерКонфигурация, основанная на статических атрибутахНе поддерживает XMLНе поддерживает технологию использования кода в качестве конфигурацииНе подерживает API, основанное на соглашенияхНе поддерживает пользовательские жизненные циклыНе поддерживает механизм перехвата |

## 3.1. ASPECT.NET

Аспектно-ориентированное программирование(АОП) – перспективное направление, которое сложилось в середине 1990-х гг. Этот подход вот уже долгие годы дополняет возможности широко используемого ООП. Суть данного подхода – поддержка разработки и модификации **сквозной функциональности** (*cross-cutting concerns*) в больших программных системах[4].

При построении какого-либо приложения бизнес-логику проекта, так или иначе, пытаются представить в виде иерархии модулей, функциональностей (то есть модулей реализующих какую-нибудь функциональную возможность), представленных в виде классов, процедур, функций и так далее. Примерами таких модулей могут стать классы, отвечающие за расчет зарплаты, расчет курсов акций и другие. Однако существуют виды функциональностей, реализация которых принципиально не осуществима в виде лишь одной иерархии взаимосвязанных модулей, а, в дополнение к ним, требует вставки в физически рассредоточенные точки программы фрагментов нового кода. Они и называются **сквозной функциональностью.** Самым простым примером может стать логгирование, которое выводит статистическую информацию об исполнении различных фрагментов кода.

Задача АОП – модуляция сквозной функциональности и её автоматизированное, безопасное и надежное добавление в целевую программу посредством **аспектов**[7]. Аспект — это реализация сквозной функциональности, выполненная в виде специального модуля, содержащего фрагменты кода (действия аспекта), активируемые в заданных точках целевой программы как часть новой функциональности[4].

**Терминология:**

**Разрез** кода целевой программы (*pointcut*) – совокупности правил поиска в ней *точек присоединения (join points)* аспекта к целевой программе для дальнейшего *внедрения(weaving)* в этих точках действий аспекта.

**Точки присоединения** (*join points*)аспекта – точки в целевой программе, где будет использована сквозная функциональность.

**Внедрение** (*weaving*) – способ использования аспекта как новой разновидности модуля.

Наиболее популярный инструмент АОП, разработанная Г. Кичалесом AspectJ[7]. В данной работе был использован разработанный под руководством Сафонова В. О. инструментарий АОП Aspect.NET[5]

## 4.1. MS Roslyn

На данный момент компиляторы превращают исходный код в объектные файлы либо сборки, являясь тем самым подобием черных ящиков. Проект Roslyn от компании Microsoft был создан для их открытия, в частности компилятора C#, предоставив возможность пользоваться API: SyntaxTree, Emit API, Symbol API, Binding & Flow Analysis API. Доступные сервисы:

 

Возможна генерация кода, анализа, оптимизации, а также поддержка сценариев и интерактивного использования C#.

Функции выделения кода и его форматирования используют синтаксические деревья, а для анализа и генерации кода используем семантическую модель, которые мы и будем рассматривать.

### SyntaxTree

Абстрактное синтаксическое дерево(*SyntaxTree*) – древовидное представление синтаксической структуры программы.

**Узлы** – конструкции языка программирования (классы, методы, операторы).

**Листья** – неделимые синтаксические конструкции (переменные, константы, ключевые слова).

Пример: 

Чтобы создать синтаксическое дерево создаем SyntaxTree: 

Так же синтаксическое дерево можно получить из конкретных выражений:

ExpressionSyntax myExpression = Syntax.ParseExpression("1+1");

StatementSyntax myStatement = Syntax.ParseStatement("for (int i = 0; i <  length; i++) { }");

Для объявления элемента в пространстве имен используется метод *Syntax.NamespaceDeclaration*:

NameSyntax namespaceName = Syntax.ParseName("Roslyn.Compilers");

NamespaceDeclarationSyntax myNamespace = Syntax.NamespaceDeclaration(

 namespaceName);

## 4.2. Элементы SyntaxTree

Существует три типа узлов синтаксического дерева, такие как: *Nodes, Tokens, Trivia*.[19] Рассмотрим их.

Каждая категория узлов представленна отдельным классом, производным от *SyntaxNode* как и обычные узлы дерева, они имеют у себя дочерние и родительские элементы, представленные в виде метода *ChildNodes* и свойства *Parent*. Каждый узел также имеет коллекцию методов *Descendant \**, таких как *DescendantNodes*, *DescendantTokens* или *DescendantTrivia*, которые представляют список всех узлов, существующих в поддереве, корневых для этого узла.

Пример использования метода *DescendantNodes*:

var tree = Roslyn.Compilers.CSharp.SyntaxTree.ParseCompilationUnit(

 @"class c1 { }");

var token = tree.Root.DescendentTokens().First();

## 4.2.1 SyntaxNode

SyntaxNodes являются одним из основных элементов синтаксических деревьев. Эти узлы представляют собой синтаксические конструкты, такие как объявления, операторы, предложения и выражения.

## 4.2.2 SyntaxToken

SyntaxToken являются элементами грамматики языка, представляя самые маленькие синтаксические фрагменты кода. Они никогда не являются родителями других Node или Trivia. SyntaxToken состоят из ключевых слов, идентификаторов, литералов и знаков препинания.

## 4.2.3 SyntaxTrivia

SyntaxTrivia представляют собой части исходного текста, которые в значительной степени незначительны для нормального понимания кода, например: пробелы, комментарии и т.д.

## 4.3 Модификация исходного кода

На базе анализа кода, можно провести изменения дерева. Например, можно изменить имя какого-либо namespace:

Получим текущий namespace:

NamespaceDeclarationSyntax oldNamespace = tree.GetRoot()

 .DescendantNodes()

 .OfType<NamespaceDeclarationSyntax>()

 .First();

Создадим новый namespace:

NamespaceDeclarationSyntax newNamespace =

 oldNamespace.WithName(“Новое имя”);

Также можно заменять узлы целиком при помощи метода ReplaceNode. Например, в предыдущем методе требуется заменить старый узел новым целиком:

CompilationUnitSyntax newRoot = tree.GetRoot()

 .ReplaceNode(oldNamespace, newNamespace);

##  4.4 Семантика

Многие типы, поля, методы и локальные переменные с одним и тем же именем могут быть распределены по всему источнику. Хотя каждое из них уникально по-разному, определение того, к какому именно идентификатору элемент относится фактически, часто требует глубокого понимания языковых правил. Поэтому MS Roslyn заводит речь о семантике.

## 4.4.1 Compilation

Компиляция представляет каждый объявленный тип, член или переменную как элемент *Symbol*. Компиляция содержит множество методов, которые помогут вам найти и связать символы, которые либо были объявлены в исходном коде, либо импортированы как метаданные из сборки. С ней можно проводить такие же операции как и с синтаксическим деревом.

Для получения компиляции нам требуются: деревья синтаксиса, ссылки на сборки и параметры компиляции.

Compilation compilation = Compilation.Create("HelloWorld")

 .AddReferences(mscorlib)

 .AddSyntaxTrees(tree);

## 4.4.2 Symbol

Символ представляет собой отдельный элемент, объявленный исходным кодом или импортированный из сборки как метаданные. Каждое пространство имен, тип, метод, свойство, поле, событие, параметр или локальная переменная представлены символом.

Разнообразие методов и свойств компиляции помогает находить эти символы. Например, вы можете найти символ для объявленного типа по его общему имени метаданных. Вы также можете получить доступ ко всей таблице символов в виде дерева, корни которого связаны с глобальным пространством имен.

Для полного понимания можно сравнить символы по своей концепции с API-интерфейсом System.Reflection, но они богаче тем, что они моделируют больше, чем просто типы. Пространства имен, локальные переменные и метки - это все символы.

Пример выделения символов из компиляции:

NamespaceSymbol globalNamespace = compilation.GlobalNamespace;

## 4.4. Semantic model

Семантическая модель представляет всю семантическую информацию для одного исходного файла. Вы можете использовать его, чтобы обнаружить следующее:

* Символы, ссылающиеся на определенное место в источнике.
* Результирующий тип любого выражения.
* Все диагностические сообщения, которые являются ошибками и предупреждениями.
* ответы на более спекулятивные вопросы.

Получить семантическую модель можно запросив её из компиляции:

SemanticModel semanticModel = compilation.GetSemanticModel(tree);

## 4.5. Workspaces

**Workspace** - это отправная точка для анализа кода и рефакторинга по всем решениям. На этом уровне API рабочей области помогает вам организовать всю информацию о проектах в решении в единую объектную модель, предлагая вам прямой доступ к моделям объектов уровня компилятора, таким как исходный текст, синтаксические деревья, семантические модели и компиляции, без необходимости анализа файлов.

Для получения Workspace используется код:

IWorkspace workspace = Workspace.LoadSolution(@"HelloWorld.sln");

ISolution solution = workspace.CurrentSolution;

## 5.1. Реализация. Схема проекта.

Для вынесения IOC-контейнеров в аспекты и реализации своих контейнеров, был взят проект электронной библиотеки Александра Полуховича[16]. Рассмотрим схему проекта:



Проект состоит из 3 частей: Model, Utils и View Model. WPF реализация не будет рассматриваться. **Model** состоит из двух интерфейсов и их реализаций – книги и библиотеки. **Utils** – вспомогательные классы, к примеру: *DelegateCommand* – для байндинга и класс *NotifyModelBase* – для добавления интерфейса в наследуемых классах *INotifyPropertyChanged*. . **ViewModel** – модель представления данного проекта. Диаграмму классов выглядит так:



Проект полностью соблюдает правила построения слабосвязанного кода и готов для внедрения в него IOC-контейнера. Был использован наиболее популярный контейнер Unity. Далее будет продемонстрирована логика программы и точки внедрения зависимостей.

Для начала рассмотрим интерфейсы классов нашей модели: книги и библиотеки:

|  |  |
| --- | --- |
| 12345678 | **public** **interface** IBook{ **string** Author { **get**; **set**; } **string** Title { **get**; **set**; } DateTime Year { **get**; **set**; } **string** SN { **get**; **set**; } **int** Count { **get**; **set**; }} |
| 123456789 | **public** **interface** ILibraryBookService{ **void** **GetData** (Action<ObservableCollection<IBook>, Exception> callback); IBook **FindBook**(IBook findBook); **void** **CreateNewBook**(); **void** **RemoveBook**(IBook book);} |

Класс LibraryBook, реализующий интерфейс IBook не имеет в себе зависимостей от других классов, однако класс библиотеки LibraryBookService будет иметь зависимости сразу в нескольких местах. В коде этого класса будет присутствовать сразу два метода внедрения зависимостей: поле класса \_books содержит в себе список книг интерфейса IBook (*Property Injection*):

|  |  |
| --- | --- |
| 1 | **private** ObservableCollection<IBook> \_books; |

И методы GetData(), FindBook(), RemoveBook(), принимающие в качестве параметров объекты типа IBook (*Method Injection*):

|  |  |
| --- | --- |
| 1234567 | **public** **void** **GetData** (Action<ObservableCollection<IBook>, Exception> callback){...}**public** IBook **FindBook**(IBook findBook){...}**public** **void** **RemoveBook**(IBook book){...} |

Не был обойден стороной и метод *Constructor Injection* он присутствует в классе MainViewModel, конструктор которого выглядит так:

|  |  |
| --- | --- |
| 12 | **public** **MainViewModel**(ILibraryBookService dataService){...}; |

Теперь, используя Unity, сконфигурируем контейнер и разрешим класс верхнего уровня MainViewModel:

|  |  |
| --- | --- |
| 12345 | **var** unityConteiner = **new** UnityContainer();unityConteiner.RegisterType<IBook,LibraryBook>();unityConteiner.RegisterType<ILibraryBookService,  LibraryBookService>();**var** model = unityConteiner.Resolve<MainViewModel>(); |

### 5.1.1. Вынесение IOC-контейнера в аспекты.

Для начала нужно понять, где в программе будут находиться точки присоединения (*join points*) аспекта. Конфигурация контейнера и разрешение класса должны происходить в самом начале программы. Поэтому для консольного приложения нужно было рассмотреть метод Main, а для WPF приложений с пользовательским интерфейсом это очевидно будет метод OnStartup()представления:

|  |  |
| --- | --- |
| 12 | **protected** **override** **void** **OnStartup**( StartupEventArgs e){...} |

Теперь можно оформить код аспекта :

|  |  |
| --- | --- |
|  1 2 3 4 5 6 7 8 910111213 | **public** **class** **SmartDIP** : Aspect{ [AspectAction("%before %call \*.**OnStartup**")] **public** **static** **void** **BeforeAction**() { **var** unityConteiner = **new** UnityContainer(); unityConteiner.RegisterType<IBook, LibraryBook>(); unityConteiner.RegisterType<ILibraryBookService, LibraryBookService>(); Program.model =unityConteiner.Resolve <MainViewModel>();} |

Тем самым в коде целевой программы не осталось ни следа контейнера, и можно спокойно использовать разрешенную модель для построения представления программы:

|  |  |
| --- | --- |
| 12 | **var** view = **new** MainWindow {DataContext = model};view.Show(); |

Либо, если это консольное приложение, можно вывести информацию о модели:

|  |  |
| --- | --- |
|  1 2 3 4 5 6 7 8 9101112131415 |  **public** **class** **Program** { **public** **static** MainViewModel model; **static** **void** **Main**(**string**[] args) { Console.WriteLine("Welcome в нашу библиотеку"); Console.Write(model.AddBookCommand .CanExecute(**new** LibraryBook {Author = "Jon Skeet" ,Title = "C# in Depth" ,Count = **3**, SN = "ISBN: 9781617291340" ,Year = **new** DateTime(**2013**, **9**, **10**) })); Console.ReadKey(); } } |

Единственное о чем должен позаботиться пользователь это создание глобальной переменной для обработки её в аспекте:

**public** **static** MainViewModel model;

Итак, вынеся весь контейнер в аспекты, можно дать характеристику контейнеру Unity. Несомненно, он хорош, и даже тот факт, что Microsoft не поддерживает его, не умаляет его значимости. Однако и у него есть минусы. Имея возможность вынести всю функциональность контейнера, было решено заменить разработанный аспект, аспектом с другим IOC-контейнером, рассмотрев, какие при этом потерпит изменения целевой код, и сравнив контейнеры между собой. Был взят аспект с контейнером Ninject внутри. Схема его API напоминает Unity:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 91011 | **public** **class** **SmartDIP** : Aspect{ [AspectAction("%before %call \*.OnStartup")] **public** **static** **void** **BeforeAction**() { IKernel ninjectKernel = **new** StandardKernel(); ninjectKernel.Bind<IBook>().To<LibraryBook>(); ninjectKernel.Bind<ILibraryBookService>() .To<LibraryBookService>(); Program.model = ninjectKernel.Get<MainViewModel>();} |

Строки 7-9 – конфигурация контейнера

Строка 10 – разрешение модели

Целевая программа с новым аспектом компилируется, запускается и даже выдает тот же результат. Тем самым получена возможность при необходимости менять IOC-контейнеры без какого-либо вмешательства в целевой код, простым нажатием кнопки в Visual Studio[2]. Действительно, данная возможность очень полезна, поскольку разные контейнеры имеют свои особенности. В данном случае мы имеем продукт Ninject, который очень удобен для построения небольших проектов, в связи с удобным API конфигурации прямо в коде. При этом он имеет очень удобный механизм управления жизненным циклом. Продемонстрируем установку для объекта логики паттерна Singleton прямо во время конфигурации контейнера строкой:

ninjectKernel.Bind<ILibraryBookService>()

 .To<LibraryBookService>()

 .**Using<Ninject.Core.Behavior.SingletonBehavior>;**

В то же время контейнер Unity больше полагается на конфигурацию в XML, а не в коде, и его механизм управления жизненным циклом объектов сложнее, из-за сложного механизма высвобождения объектов[9]. При этом если посчитать время выполнения этапа конфигурации с помощью сервиса StopWatch мы получим следующие результаты:

Для Unity: 

Для Ninject:

Unity заметно выигрывает по времени, ну а если посмотреть результаты работы этих контейнеров на проектах промышленных масштабов [13], то можно увидеть как Ninject начинает опаздывать в разы от своего оппонента:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Container** | **Singleton** | **Transient** | **Combined** | **Complex** | **Property** | **Generics** |
| [**Ninject 3.2.2.0**](http://ninject.org/) | **72594486** | **2427615296** | **6921439610** | **191033\*117563** | **165177106569** | **6734742019** |
| [**Unity 3.5.1404.0**](http://msdn.microsoft.com/unity) | **28733812** | **71823038** | **152948012** | **3911621296** | **4023421537** |  |

В таблице приведено время работы метода Resolve() контейнеров на объектах различных интерфейсов и паттернов в миллисекундах. Наглядно видно многократное преимущество в скорости контейнера Unity.

## 5.2 Анализ и рефакторинг посредством MS Roslyn

Недостатком ручного рефакторинга кода с помощью ASPECT.NET, является тот факт, что при условиях постоянного изменения целевого проекта (смены ключевых имен модулей, замена типа IOC-контейнера), может возникнуть необходимость добавления нового кода в аспект. Это может сильно усложнить проектирование кода. Поэтому далее будет продемонстрированно как можно автоматизировать данный процесс с помощью MS Roslyn.

В данном случае аспект полностью лишается какого-либо кода, связанного с инициализацией IOC-контейнера. Вместо него аспект будет состоять лишь из кода MS Roslyn, который будет генерировать вызовы соответствующих настроек.

Как и при ручном рефакторинге, точки присоединения (*join points*) аспекта расположены в самом начале программы. Для консольного приложения это метод Main, а для WPF приложений - метод OnStartup()представления.

Настройка аспекта:

|  |  |
| --- | --- |
|  1 2 3 4 5 6 7 | **public** **class** **SmartDIP** : Aspect{ [AspectAction("%before %call \*.**OnStartup**")] **public** **static** **void** **BeforeAction**() {  } |

Теперь осталось найти вхождения всех инициализаций IOC-контейнеров в целевую программу. Для этого достаточно отыскать с помощью MS Roslyn все методы Resolve(), отвечающие за настройку контейнеров.

Задача состоит в анализе исходного кода, для начала нужно получить синтаксическое дерево проекта. После установки проекта в visual studio получаем функционал для наглядной демонстрации SyntaxTree.



Указав на искомую строку кода, получаем информацию об IOC-контейнере, который хотим извлечь.

Теперь построим дерево программным методом.

Для построения синтаксического дерева понадобится текст целевого проекта – в данном случае это *SimpleExample.sln*. Его код можно выделить, используя Workspace.

Строим Workspace, предварительно добавив ссылку на Roslyn.Services:

|  |  |
| --- | --- |
| 1234567 | tring solutionPath = @"C:\Users\Сергей  \Downloads\Материалы для-161 \Diplom\UnityNinjectInAspect \SimpleExample.sln";**var** workspace = MSBuildWorkspace.Create();**var** solution = workspace .OpenSolutionAsync(solutionPath).Result; |

Теперь можем получить интересующий нас текст. Пройдем по нашему решению. В каждом решении нужно выделить проекты, из каждого проекта – документы.

|  |  |
| --- | --- |
| 12345678 | **foreach** (var project **in** solution.Projects) { **foreach** (var document **in**  project.Documents)  { //код обработки документа }} |

Получаем множество документов, находящихся в целевом решении.

Теперь, для каждого текста, можно построить искомое синтаксическое дерево:

|  |  |
| --- | --- |
| 1 | SyntaxTree tree =document.GetSyntaxTreeAsync().Result; |

Имея дерево, построим компиляцию и семантическую модель:

|  |  |
| --- | --- |
|  1 2 3 4 5 6 7 8 910 | MetadataReference mscorlib = MetadataReference .CreateFromFile("mscorlib");**var** compilation = CSharpCompilation .Create("SimpleExample") .AddReferences(mscorlib) .AddSyntaxTrees(tree);SemanticModel semanticModel = compilation .GetSemanticModel(tree); |

Весь код MS Roslyn пишется прямо в аспекте.

Теперь у нас есть вся аналитическая информация о целевом проекте.

По данной информации нужно найти инициализации IOC-контейнеров. Инициализации контейнеров осуществляются в теле метода **Resolve()**, о котором уже говорилось выше. Найдем этот метод.

Для этого создадим класс *MyWolker* являющийся Visiter’ом, проходящим по нашему синтаксическому дереву. Перезапишем метод *VisitMethodDeclaration* класса *MyWalker*, указывая условия прохождения по всем методам документа.

|  |  |
| --- | --- |
|  1 2 3 4 5 6 7 8 91011121314151617181920212223242526272829 | **var** root = tree.GetRoot();**var** walker = **new** MyWalker(semanticModel);walker.Visit(root);**var** result = walker.sb.ToString();**public** **class** **MyWalker** : CSharpSyntaxWalker{ SemanticModel model; **public** **MyWalker**(SemanticModel model) : **base**(SyntaxWalkerDepth.Token) { **this**.model = model; } **public** StringBuilder sb =**new** StringBuilder(); **public** **override** **void** **VisitMethodDeclaration** (MethodDeclarationSyntax node) { SymbolInfo info =model.GetSymbolInfo(node); **if**(info.GetMembers("Resolve").Any()) {  sb.Append(node.ToFullString());  **base**.VisitMethodDeclaration(node); } }} |

Здесь можно увидеть как благодаря методу *GetSymbolInfo(node)* из семантического дерева берется информация о текущем методе.

Таким образом с помощью Visiter находим методы Resolve() и записываем их код в текстовую переменную *result.*

Получив текст искомого метода, его можно запустить в нашем аспекте при помощи скрипта Roslyn. Обозначив пространство имен Roslyn.Scripting, запустим наш отрывок:

|  |  |
| --- | --- |
| 123 | ScriptEngine engine = **new** ScriptEngine();Session session = Session.Create();engine.Execute(result, session); |

Полный код аспекта можно посмотреть в приложении.

Так при помощи MS Roslyn мы динамически вынесли IOC-контейнеры целевого проекта в аспекты.

Метрики аспекта ручного вынесения:



Метрики аспекта автоматического вынесения:



Может показаться, что код аспекта при ручном рефакторинге более компактный и обладает лучшими метриками, по сравнению с автоматической обработкой, но это не так. Увеличение кода до промышленных масштабов очевидно приведет к увеличению кода аспекта за счет нюансов в виде большого количества зависимостей, различных типов IOC-контейнеров и т.д., когда код Roslyn всегда будет выглядеть одинаково.

## 6. Заключение

### 6.1. Результаты данной работы

В ходе выполнения данной работы были получены перечисленные ниже результаты:

* Изучена предметная область – принцип внедрения зависимостей, IOC-контейнеры и их реализации, MS Roslyn.
* Был произведен анализ контейнеров Unity и Ninject.
* Используя технологию Aspect.NET, IOC-контейнеры были вынесены в аспекты, тем самым целевой код полностью был избавлен от швов, причем результаты сборки не изменились.
* Используя MS Roslyn, была проведена работа по анализу кода и автоматическому поиску IOC-контейнеров в целевой программе

### 6.2. Направление дальнейших исследований

В дальнейшем стоит провести сравнительный анализ Aspect.NET и IOC-контейнеров и попробовать реализовать в Aspect.NET функционал контейнеров. Также стоит вести работы по расширению возможностей Aspect.NET. Также в связи с увеличением возможностей MS Roslyn провести работу по усовершенствованию общения целевого проекта с ASPECT.NET в будущих реализациях продукта.

Исходный код разработанных программ можно скачать здесь - <https://github.com/qwertin149/Diplom>

## 7. Список литературы

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидс Д*. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. Издательство «Питер». 1994

[2] *Зотов М*. Реализация надстройки MS VS 2012 для поддержки системы аспектно-ориентированного программирования Aspect.NET. СПИСОК-2014. Материалы Всероссийской научной конференции по проблемам информатики. Санкт-Петербург, 2014.

[3] *Ларман К*. Применение UML и шаблонов проектирования. Второе издание. Издательский дом «Вильямс».2004

[4] *Cафонов В. О*. Аспектно-ориентированное программирование. Издательский дом Санкт-Петербургского государственного университета. 2011

[5] *Сафонов В.О*. Aspect.NET — a new approach to aspect-oriented programming // .NET Developers Journal. 2003.

[6] *Симан М*. Внедрение зависимостей в .NET. Издательство «Питер». 2013

[7] *Laddad R*. AspectJ in action. Practical aspect-oriented programming. Manning, 2003.

[8] Документация Microsoft: <https://msdn.microsoft.com/library>//Проверено 20.05.2017

[9] Сайт разработчиков Unity: <https://msdn.microsoft.com/en-us/library/ff647202.aspx> //Проверено 20.05.2017

[10] Сайт разработчиков Ninject: [www.ninject.org/](http://www.ninject.org/) //Проверено 20.05.2017

[11] Сайт разработчиков MEF: <https://msdn.microsoft.com/en-us/magazine/gg650670.aspx> //Проверено 20.05.2017

[12] Форум с обсуждениями контейнера Autofac: <http://www.codeproject.com/Articles/25380/Dependency-Injection-with-Autofac>//Проверено 20.05.2017

[13] Форум, посвященный DI-паттерну: <http://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison> //Проверено 20.05.2017

[14] Форум, посвященный Spring.NET: <http://www.springframework.net/doc-latest/reference/html/objects.html>//Проверено 20.05.2017

[15] *Мурадов М*.: Личный блог: <http://blog.muradovm.com/2012/01/blog-post.html>//Проверено 20.05.2017

[16] *Полухович А*. Личный блог: <http://sonyks2007.blogspot.ru>//Проверено 20.05.2017

[17] *Тепляков С*. Личный блог: <http://sergeyteplyakov.blogspot.ru>//Проверено 20.05.2017

[18] *Karen Ng*., *Warren,MS*., *Peter Golde*., *Anders Hejlsberg.* The Roslyn Project Exposing the C# and VB compiler’s code analysis. Microsoft Corporation. 2012

[19] *Тепляков С*. A simple analyzer using Roslyn. Личный блог: <http://sergeyteplyakov.blogspot.ru>//Проверено 20.05.2017

## 8. Приложение

Далее представлен код аспекта:

1. Аспект с вынесенным IOC-контейнером Unity:

|  |  |
| --- | --- |
|  1 2 3 4 5 6 7 8 910111213 | **public** **class** **SmartDIP** : Aspect{ [AspectAction("%before %call \*.OnStartUp")] **public** **static** **void** **BeforeAction**() { **var** unityConteiner = **new** UnityContainer(); unityConteiner.RegisterType<IBook, LibraryBook>(); unityConteiner.RegisterType<ILibraryBookService,  LibraryBookService>(); Program.model =  unityConteiner.Resolve<MainViewModel>(); }} |

1. Аспект с вынесенным IOC-контейнером Ninject:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 910111213 | **public** **class** **SmartDIP** : Aspect{ [AspectAction("%before %call \*.OnStartUp")] **public** **static** **void** **BeforeAction**() { **IKernel** ninjectKernel = **new** StandartKernel(); ninjectKernel.Bind<IBook>().To<LibraryBook>();  ninjectKernel.Bind<ILibraryBookService>() .To<LibraryBookService>();  Program.model =  ninjectKernel.Get<MainViewModel>(); }} |

1. Аспект, реализующий функциональность IOC-контейнера (упрощенный вариант для метода *Method Injection*):

|  |  |
| --- | --- |
| 123456 | [AspectAction("%before %call \*GetAllDiscount(..)")] **public** **static** **void** **BeforeAction**(){ Program.Storage = **new** Warehouse(); Program.Scheme = **new** SimpleSсheme();} |

1. Аспект, реализующий функциональность IOC-контейнера (вариант с механизмом отражений для метода *Constructor Injection*):

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 910111213141516171819 | [AspectAction("%instead %call \*Create(..)  && %args(..)")]**public** **static** ProductService **InsteadAction** (**string** Storage, **string** Calculator){ Type lentaType = System.Type.GetType ("SimpleExample.DIP."+Storage); IProductStorage lenta = (IProductStorage) lentaType.GetConstructors()[**0**].Invoke(**null**); Type schemeType = System.Type.GetType ("SimpleExample.DIP." + Calculator); IDiscountCalculator calculator=(IDiscountCalculator) schemeType.GetConstructors()[**0**].Invoke(**null**); ProductService service =  **new** **ProductService**(lenta, calculator); **return** service;} |

1. Автоматический рефакторинг при помощи MS Roslyn:

|  |  |
| --- | --- |
|  1 2 3 4 5 6 7 8 91011121314151617181920212223242526272829303132333435363738394041424344454647484950515253545556575859606162636465666768697071 | **namespace** **HelloAspect**{ **public** **class** **SmartDIP** : Aspect { [AspectAction("%before %call \*.writeline")] **public** **static** **void** **BeforeAction**() { **string** solutionPath = @"C:\Users\Сергей \Downloads\Материалы-для-161\Diplom \UnityNinjectInAspect\SimpleExample.sln"; **var** workspace = MSBuildWorkspace.Create(); **var** solution = workspace .OpenSolutionAsync(solutionPath).Result; **foreach** (**var** project **in** solution.Projects) { **foreach** (**var** document **in** project.Documents) { SyntaxTree tree = document.GetSyntaxTreeAsync().Result; MetadataReference mscorlib = MetadataReference.CreateFromFile("mscorlib"); **var** compilation = CSharpCompilation.Create("SimpleExample") .AddReferences(mscorlib) .AddSyntaxTrees(tree); SemanticModel semanticModel = compilation .GetSemanticModel(tree); **var** root = tree.GetRoot(); **var** walker = **new** MyWalker(semanticModel); walker.Visit(root); **var** result = walker.sb.ToString(); ScriptEngine engine = **new** ScriptEngine(); Session session = Session.Create(); engine.Execute(result, session); } } } **public** **class** **MyWalker** : CSharpSyntaxWalker { SemanticModel model; **public** **MyWalker**(SemanticModel model) : **base**(SyntaxWalkerDepth.Token) { **this**.model = model; } **public** StringBuilder sb = **new** StringBuilder(); **public** **override** **void** **VisitMethodDeclaration**(MethodDeclarationSyntax node) { SymbolInfo info = model.GetSymbolInfo(node); **if** (info.GetMembers("Resolve").Any()) { sb.Append(node.ToFullString()); **base**.VisitMethodDeclaration(node); } } }  }} |