Этот пример показывает, как для настройки локатора сервисов используется статический метод `Register`. Если, как показано в примере, это сделано до создания `ProductService` этим сервисом, то для работы с хранилищем `ProductRepository` используется настроенная заглушка. В готовом приложении локатор сервисов будет настроен в корне композиции на подходящую реализацию `ProductRepository`.

Этот способ определения зависимостей из класса `ProductService` действительно работает при условии, что единственным критерием успеха является возможность использования зависимости и ее замены по желанию. Но у него есть серьезные недостатки.

5.2.2. Анализ антипаттерна «Локатор сервисов»

«Локатор сервисов» — весьма опасный антипаттерн, потому что он почти всегда работоспособен. Зависимости можно найти из потребляющих классов, и эти зависимости можно заменить другими реализациями — даже тестовыми дубликатами из модульных тестов. Применяя аналитическую модель, изложенную в главе 1, в целях вычисления тех случаев, когда локатор сервисов может соответствовать преимуществам модульной конструкции приложения, обнаружится, что он подходит в большинстве случаев.

- Можно поддерживать позднее связывание путем изменения регистрации.
- Можно разрабатывать код параллельно, поскольку программирование ведется с прицелом на использование интерфейсов, заменяя модули по своему желанию.
- Можно добиться весьма качественного разделения обязанностей, поэтому ничто не останавливает вас от написания сопровождаемого кода, просто это будет сложнее сделать.
- Зависимости можно заменять тестовыми дубликатами, обеспечивая тем самым тестируемость.

Есть только одна область, в которой локатор сервисов не соответствует требованиям, и к этому нельзя относиться легкомысленно.

Негативные эффекты от применения антипаттерна «Локатор сервисов»

Основная проблема при использовании локатора сервисов состоит в том, что он влияет на возможность повторного использования классов-потребителей. Это проявляется двояко:

- класс тащит за собой локатор сервисов как избыточную зависимость;
- класс скрывает очевидность своих зависимостей.

Посмотрим сначала на показанный на рис. 5.4 граф зависимостей для `ProductService` из примера, приведенного в подразделе 5.2.1.

Кроме ожидаемой ссылки на `IProductRepository`, `ProductService` также зависит от класса `Locator`. Это означает, что для повторного использования класса

`ProductService` необходимо снова распространить не только его и относящуюся к нему зависимость `IProductRepository`, но и зависимость от класса `Locator`, существующую только по техническим причинам. Если класс `Locator` определен в другом модуле, не в том, что `ProductService` и `IProductRepository`, то новое приложение, собирающееся использовать `ProductService`, также должно принять этот модуль.

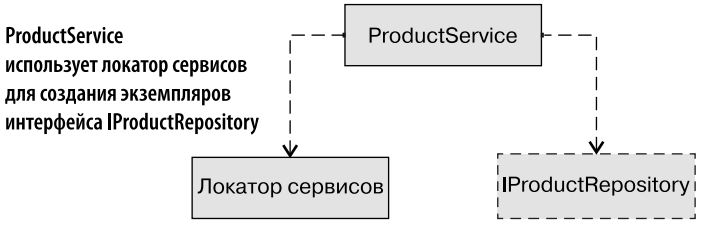

Рис. 5.4. Граф зависимостей для ProductService

Возможно, эту дополнительную зависимость от класса `Locator` и можно было бы допустить, если бы он действительно был необходим для работы технологии DI. Мы бы восприняли его как налог, уплачиваемый за получение преимуществ. Но имеются и более удачные и доступные варианты (например, внедрение через конструктор), поэтому данная зависимость представляется избыточной. Более того, в поле зрения разработчиков, желающих воспользоваться классом `ProductService`, ни эта избыточна, зависимость, ни связанное с ним дополнение в виде `IProductRepository` явным образом не видны. На рис. 5.5 показано, что среда Visual Studio не предлагает никаких подсказок по использованию этого класса.

Рис. 5.5. Единственное, о чем может сообщить принадлежащая среде Visual Studio служба IntelliSense о классе ProductService, это то, что у него имеется конструктор без параметров. Все его зависимости остаются невидимыми

Если нужно создать новый экземпляр класса `ProductService`, среда Visual Studio может только сообщить вам, что у класса имеется конструктор без параметров. Но если впоследствии вы попытаетесь запустить код, то при отсутствии регистрации экземпляра `IProductRepository` с помощью класса `Locator` будет получена ошибка времени выполнения. Это может произойти, если вы плохо знаете устройство класса `ProductService`.

ПРИМЕЧАНИЕ

Представьте себе, что создаваемый вами код поставляется в непонятном .dll-файле без документации. Просто ли будет кому-либо им воспользоваться? Можно разработать практически самодокументируемые API, и, хотя это под силу только опытным программистам, это весьма достойная цель. Проблема, связанная с локатором сервисов, состоит в том, что любой использующий его компонент дезориентирует разработчиков относительно своего уровня сложности. Через открытый API он выглядит простым, но на самом деле является сложным, что не может быть обнаружено до первой попытки его запуска на выполнение.

Класс `ProductService` далек от самодокументирования: до того, как он заработает, абсолютно невозможно сказать что-либо о том, какие зависимости должны присутствовать. Фактически в будущих версиях разработчики `ProductService` могут даже принять решение о добавлении дополнительных зависимостей. Это будет означать, что код, работающий для текущей версии, в будущей версии может дать сбой, а предупреждающей ошибки компилятора получено не будет. Локатор сервисов облегчает непреднамеренное внесение изменений, не поддерживающих обратной совместимости.

Использование обобщений может навести на мысль, что локатор сервисов обладает строгой типизацией. Но даже API, подобный показанному в листинге 5.7, имеет слабую типизацию, поскольку может быть запрошен любой тип. Возможность компиляции кода, вызывающего метод `GetService<T>`, не дает вам никаких гарантий, что он не станет направо и налево выдавать исключения в процессе выполнения приложения.

В ходе модульного тестирования возникает еще одна проблема, связанная с тем, что тестовый дубликат, зарегистрированный в одном наборе тестовых данных, приведет к созданию неудачного кода под названием «Взаимозависимые тесты» (Interdependent Tests), поскольку он останется в памяти при выполнении следующего набора тестовых данных. Поэтому необходимо после каждого теста производить перенастройку тестовых установок (Fixture Teardown), вызвав метод `Locator.Reset()`[1]. Нужно не забыть это сделать самостоятельно, на чем легко можно оступиться.

Дело совсем не в механике

Хотя локатор сервисов принимает различные формы и обличия, его общая сигнатура выглядит примерно так:

```
public T Resolve<T>()
```

Нетрудно предположить, что каждый API с этой сигнатурой является локатором сервисов, но это не так. На самом деле это та самая сигнатура, которая демонстрируется большинством DI-контейнеров. Как локатор сервисов, он определяется не статической структурой API, а той ролью, которую API играет в приложении.

Важный аспект антипаттерна «Локатор сервисов» заключается в том, что компоненты приложения запрашивают зависимости вместо статического их объявления через свой конструктор. Как уже ранее объяснялось, это сопряжено с массой недостатков. Но когда код, являющийся частью корня композиции, запрашивает зависимости, эти недостатки не проявляются.

Поскольку корень композиции уже зависит от всего остального, имеющегося в системе (о чем уже говорилось в разделе 4.1), тащить за собой дополнительную зависимость он просто не может. Ему уже по определению известно о каждой зависимости.

[1] Дополнительные сведения о перенастройке тестовых установок можно найти в издании: *Meszaros G.* xUnit Test Patterns. — P. 100.

Скрывать свои зависимости корень композиции не может — от кого же ему их скрывать? Его роль заключается в построении графа объектов; ему не нужно предоставлять эти зависимости.

Запрос зависимостей, даже если он делается через DI-контейнер, при неправильном применении становится локатором сервисов. Когда код приложения (в отличие от кода инфраструктуры) активно запрашивает сервис для получения необходимых зависимостей, он становится локатором сервисов.

ВНИМАНИЕ

DI-контейнер, заключенный в корень композиции, локатором сервисов не является. Он представляет собой компонент инфраструктуры.

Использование локатора сервисов может показаться совершенно безобидным, но оно может привести к всевозможным неприятным ошибкам времени выполнения. Как избежать этих проблем? Когда принимается решение об избавлении от локатора сервисов, нужно найти соответствующий способ. Подходом по умолчанию должно стать внедрение через конструктор, если только не станет более подходящим какой-нибудь другой паттерн DI из главы 4.

Рефакторинг антипаттерна «Локатор сервисов» и переход к применению технологии DI

Поскольку при внедрении через конструктор происходит статическое объявления зависимостей класса, то использование чистой технологии DI приведет к ошибке компиляции. А вот при использовании DI-контейнера утрачивается возможность проверки правильности кода в ходе компиляции. Но статически объявленные зависимости класса все же гарантируют возможность проверки правильности графа объектов вашего приложения путем запроса к контейнеру на создание для вас всех графов объектов. Это можно сделать при запуске приложения или же в виде составной части модульного или интеграционного теста.

В некоторых DI-контейнерах сделан даже еще один шаг вперед, позволяющий проводить более сложный анализ DI-конфигурации. Он разрешает обнаруживать все разновидности самых распространенных подводных камней. Локатор сервисов будет полностью невидим DI-контейнеру, не позволяя тем самым проводить подобные проверки от вашего имени.

Во многих случаях у класса-потребителя локатора сервисов могут быть вызовы к нему, распространяемые по всей кодовой базе. В таких случаях он действует в качестве замены инструкции `new`. Когда такое происходит, первым шагом по переработке является объединение создания каждой зависимости в одном методе.

При отсутствии у компонента поля для хранения экземпляра зависимости вы можете ввести это поле и обеспечить его использование всем остальным кодом, когда он является потребителем зависимости. Поле нужно снабдить пометкой `readonly`,

исключив возможности его изменения за пределами конструктора. Подобные действия вынуждают вас присваивать полю значение из конструктора, используя локатор сервисов. Теперь можно ввести параметр конструктора, присваивающий полю значение вместо «Локатора сервисов», который затем может быть удален.

ПРИМЕЧАНИЕ

Введение в конструктор параметра зависимости, скорее всего, приведет в негодность уже имеющихся потребителей, поэтому лучше начать с самых верхних классов и спускаться вниз по графу зависимостей.

Переработка класса, применяющего локатор сервисов, похожа на переделку класса, использующего антипаттерн «Диктатор». Дополнительные заметки по переработке реализации антипаттерна «Диктатор» в применение технологии DI содержатся в подразделе 5.1.4.

На первый взгляд локатор сервисов может показаться вполне допустимым паттерном DI, но не стоит обманываться: он может явным образом решить задачу слабой связанности, но при этом принести в жертву другие решаемые вопросы. DI-паттерны, представленные в главе 4, предлагают более подходящие варианты с меньшим количеством недостатков. Это справедливо как для антипаттерна «Локатор сервисов», так и для других антипаттернов, представленных в этой главе. Несмотря на то что они отличаются друг от друга, все эти антипаттерны имеют одну общую особенность: связанные с ними проблемы могут быть успешно решены путем использования одного из DI-паттернов, рассмотренных в главе 4.

5.3. Антипаттерн «Окружающий контекст»

Родственным локатору сервисов является антипаттерн «Окружающий контекст». Если локатор сервисов разрешает глобальный доступ к неограниченному набору зависимостей, то антипаттерн «Окружающий контекст» открывает доступ через статический метод доступа к одной строго типизированной зависимости.

ОПРЕДЕЛЕНИЕ

Антипаттерн «Окружающий контекст» с помощью статических компонентов класса предоставляет коду приложения, находящемуся вне корня композиции, глобальный доступ к нестабильной зависимости или к ее поведению.

В листинге 5.9 антипаттерн «Окружающий контекст» показан в действии.

В этом примере `ITimeProvider` представляет собой абстракцию, позволяющую извлекать текущее время системы. Поскольку может потребоваться повлиять на восприятие времени приложением (например, для тестирования), вызывать напрямую `DateTime.Now` нежелательно. Вместо того чтобы позволить потребителям вызывать `DateTime.Now` напрямую, более удачным решением может стать маскировка доступа к `DateTime.Now` за абстракцией. Заманчиво, однако, предоставить потребителям доступ к реализации этой абстракции по умолчанию через статическое свойство

или метод. Показанное в листинге 5.9 свойство `Current` открывает доступ к такой реализации `ITimeProvider`.

Листинг 5.9. Использование антипаттерна «Окружающий контекст»

Статическое свойство Current представляет окружающий контекст, открывающий доступ к экземпляру ITimeProvider. В результате зависимость ITimeProvider становится скрытой, а тестирование затрудняется

```
public string GetWelcomeMessage()
{
    ITimeProvider provider = TimeProvider.Current;
    DateTime now = provider.Now;

    string partOfDay = now.Hour < 6 ? "night" : "day";

    return string.Format("Good {0}.", partOfDay);
}
```

Окружающий контекст структурно похож на паттерн «Одиночка» (Singleton)[1]. Оба они открывают доступ к зависимости путем использования компонентов статического класса. Разница в том, что окружающий контекст позволяет своей зависимости изменяться, а паттерн «Одиночка» гарантирует абсолютную неизменчивость своего единственного экземпляра.

ПРИМЕЧАНИЕ

Паттерн «Одиночка» должен использоваться либо из корня композиции, либо когда зависимость является стабильной. С другой стороны, когда паттерн «Одиночка» используется для предоставления приложению глобального доступа к нестабильной зависимости, последствия его применения аналогичны, как говорится в подразделе 5.3.3, тем, что возникают при использовании антипаттерна «Окружающий контекст».

Доступ к текущему времени системы является общей потребностью. Немного тщательнее присмотримся к примеру `ITimeProvider`.

5.3.1. Пример: получение данных о времени через «Окружающий контекст»

Потребность в получении контроля над временем возникает по многим причинам. У большинства приложений есть бизнес-логика, зависящая от времени или хода времени. В предыдущем примере был показан простой случай вывода приветственного сообщения на основе текущего времени. Два других примера включают в себя следующее.

- *Вычисление стоимости на основе дня недели.* В некоторых бизнес-сферах вполне естественно для клиентов платить больше за обслуживание в выходные дни.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 157.

- *Отправка пользователям уведомлений с использованием разных каналов связи на основе времени суток.* Например, в рабочие часы может быть сподручнее отправлять уведомления по электронной почте, а в остальное время лучше отправлять их в виде СМС.

Поскольку востребованность работы со временем крайне высока, у разработчиков зачастую возникает потребность в упрощении доступа к такой нестабильной зависимости путем использования антипаттерна «Окружающий контекст». Пример абстракции `ITimeProvider` показан в листинге 5.10.

Листинг 5.10. Абстракция ITimeProvider

```
public interface ITimeProvider
{
    DateTime Now { get; }
}
```

Позволяет потребителям узнать текущее время системы

В листинге 5.11 показана упрощенная реализация класса `TimeProvider` для этой абстракции `ITimeProvider`.

Листинг 5.11. Реализация антипаттерна «Окружающий контекст» в виде класса TimeProvider

```
public static class TimeProvider
{
    private static ITimeProvider current =
        new DefaultTimeProvider();

    public static ITimeProvider Current
    {
        get { return current; }
        set { current = value; }
    }

    private class DefaultTimeProvider : ITimeProvider
    {
        public DateTime Now { get { return DateTime.Now; } }
    }
}
```

Статический класс, отрывающий глобальный доступ к сконфигурированной реализации ITimeProvider

Инициализация локальной реализации по умолчанию, использующей реальные системные часы

Статическое свойство, открывающее глобальный доступ по чтению и записи к нестабильной зависимости ITimeProvider

Реализация по умолчанию, использующая реальные системные часы

Используя реализацию `TimeProvider`, можно проводить модульное тестирование ранее определенного метода `GetWelcomeMessage`. Такой тест показан в листинге 5.12.

Это один из вариантов антипаттерна «Окружающий контекст». Другими часто встречающимися вариантами являются:

- *окружающий контекст, позволяющий потребителям воспользоваться поведением глобально сконфигурированной зависимости.* Возвращаясь к предыдущему примеру, `TimeProvider` может предоставить потребителям статический метод `GetCurrentTime`, скрывающий используемую зависимость за счет ее внутреннего вызова;

- *окружающий контекст, объединяющий статический метод доступа с интерфейсом в единую абстракцию.* В отношении предыдущего примера это означало бы наличие единственного основного класса `TimeProvider`, содержащего как свойство экземпляра `Now`, так и статическое свойство `Current`;
- *окружающий контекст, где вместо абстракции, определяемой пользователем, используются делегаты.* Вместо того чтобы располагать интерфейсом `ITimeProvider`, вполне достаточное описание которого содержится в нем самом, можно добиться того же самого, воспользовавшись делегатом `Func<DateTime>`.

Листинг 5.12. Модульный тест, зависящий от антипаттерна «Окружающий контекст»

```
[Fact]
public void SaysGoodDayDuringDayTime()
{
    // Подготовка
    DateTime dayTime = DateTime.Parse("20190101 6:00");

    var stub = new TimeProviderStub { Now = dayTime };

    TimeProvider.Current = stub;

    var sut = new WelcomeMessageGenerator();

    // Действие
    string actualMessage = sut.GetWelcomeMessage();

    // Утверждение
    Assert.Equal(expected: "Good day.", actual: actualMessage);
}
```

Замена реализации по умолчанию заглушкой, которая всегда возвращает указанное значение dayTime

API WelcomeMessageGenerator проявляет скрытность, поскольку его конструктор скрывает факт того, что ITimeProvider является обязательной зависимостью

Между TimeProvider.Current и GetWelcomeMessage возникает временная связанность

Окружающий контекст может возникать во многих обличиях и реализациях. Опять же предостережение насчет окружающего контекста состоит в том, что он обеспечивает либо прямой, либо косвенный доступ к нестабильной зависимости с помощью некоторого статического компонента класса. Прежде чем провести анализ и оценку возможных способов устранения проблем, вызванных окружающим контекстом, рассмотрим еще один весьма распространенный пример окружающего контекста.

5.3.2. Пример: ведение регистрационных записей с помощью окружающего контекста

Еще один весьма распространенный случай, когда разработчики стремятся сократить путь, попадая в ловушку, касается применения в приложении регистрационных записей. Любое реальное приложение требует возможности вести запись информации

об ошибках и других необычных условиях в файл или другой источник для последующего анализа. Многие разработчики чувствуют, что ведение регистрационных записей является особой деятельностью, заслуживающей «отступления от правил». Код, подобный показанному в листинге 5.13, можно найти даже в кодовой базе разработчиков, достаточно хорошо знакомых с технологией DI.

Листинг 5.13. Применение окружающего контекста при ведении журналов

```
public class MessageGenerator
{
    private static readonly ILog Logger =
        LogManager.GetLogger(typeof(MessageGenerator));

    public string GetWelcomeMessage()
    {
        Logger.Info("GetWelcomeMessage called.");

        return string.Format(
            "Hello. Current time is: {0}.", DateTime.Now);
    }
}
```

Получение нестабильной зависимости ILog с помощью статического окружающего контекста LogManager и сохранение ее в приватном статическом поле. Это скрывает зависимость и затрудняет тестирование класса MessageGenerator

Использование поля Logger для регистрации каждого вызова метода

Широкая распространенность антипаттерна «Окружающий контекст» во многих приложениях, занимающихся ведением регистрационных записей, обусловливается несколькими причинами. Во-первых, код, похожий на показанный в листинге 5.13, обычно фигурирует в качестве первого примера, показываемого в документации библиотек со средствами журналирования. Разработчики, не разобравшись, копируют эти примеры. Винить их не за что, ведь они предполагают, что разработчикам библиотек известны наиболее рациональные приемы программирования, которыми они готовы поделиться. К сожалению, бывает и иначе. Примеры в документации даются для простоты понимания, а не для распространения передового опыта, даже если он и имеется в арсенале разработчиков библиотек.

Кроме того, антипаттерн «Окружающий контекст» зачастую применяется разработчиками для систем регистрации из-за того, что им нужно входить в своем приложении практически в каждый класс. Внедрение его в конструктор может легко привести к созданию конструкторов с переизбытком зависимостей. В результате получится рассматриваемый в главе 6 проблемный код, который называется *избыточным внедрением через конструктор* (Constructor Over-injection).

Джефф Этвуд (Jeff Atwood) еще в 2008 году опубликовал в блоге большую статью, посвященную опасности регистрирования (https://blog.codinghorror.com/the-problem-with-logging/). Перечислим некоторые из его аргументов.

- Регистрирование ведет к увеличению объема кода, из-за чего становится труднее разобраться в коде самого приложения.

- При регистрировании приходится неизменно чем-то жертвовать, и в большинстве случаев оно предполагает постоянный сброс данных на диск.
- Чем больше объем регистрирования, тем труднее поиск нужной информации.
- Если данные стоит сохранять в регистрационном файле, то их стоит показать в пользовательском интерфейсе.

Во время своей работы над Stack Overflow Джефф удалил практически все регистрирование, оставив только запись необработанных исключений. Если возникла ошибка, должно быть выдано исключение.

Всецело соглашаясь с анализом Джеффа, нам все же хотелось бы подойти к данному вопросу с точки зрения проектирования. По нашему мнению, при качественно спроектированном приложении можно было бы воспользоваться регистрированием взаимоотношений основных компонентов, не засоряя им всю кодовую базу. Более подробно разработка такого приложения рассматривается в главе 10.

ПРИМЕЧАНИЕ

Наши рассуждения ни в коем случае не нужно воспринимать как полный отказ от регистрирования. Ведение регистрационных записей является жизненно важной частью любого приложения, что справедливо и для тех приложений, которые мы создаем. Сказанное нами нужно воспринимать так, что ваше приложение должно разрабатываться с прицелом на то, что журналированием будут охвачены только несколько классов вашей системы. Если за ведение регистрационных записей будет отвечать основная часть компонентов вашего приложения, сопровождать код станет сложнее.

Существует множество других примеров использования антипаттерна «Окружающий контекст», но эти два примера настолько распространены, что в компаниях, обращавшихся к нам за консультацией, нам приходилось сталкиваться с ними несчетное количество раз. (Мы даже винили себя за то, что в прошлом сами внедряли применение антипаттерна «Окружающий контекст» в практику программирования.) После рассмотрения двух самых распространенных примеров применения антипаттерна «Окружающий контекст» в следующем подразделе предстоит понять суть создаваемой им проблемы и способы, позволяющие с ней справиться.

5.3.3. Анализ антипаттерна «Окружающий контекст»

Антипаттерн «Окружающий контекст» обычно встречается, когда у разработчиков появляется повсеместно используемая сквозная функциональность в виде нестабильной зависимости. Такая общедоступность наводит разработчиков на мысль об оправданности их отхода от применения внедрения зависимостей через конструктор. Она позволяет им скрыть зависимости и избавиться от необходимости добавления зависимости к множеству конструкторов в их приложении.

Негативные последствия применения антипаттерна «Окружающий контекст»

Проблемы при использовании антипаттерна «Окружающий контекст» перекликаются с проблемами, связанными с использованием антипаттерна «Локатор сервисов». Вот основные из них.

- Зависимость имеет скрытую форму.
- Тестирование усложняется.
- Затрудняется изменение зависимости, основанной на ее контексте.
- Между инициализацией зависимости и ее использованием возникает временная связанность.

Когда зависимость скрывается за счет разрешения к ней глобального доступа через использование антипаттерна «Окружающий контекст», становится проще утаить факт наличия у класса слишком большого числа зависимостей. Это связано с использованием проблемного кода, который называется избыточным внедрением через конструктор и зачастую служит признаком нарушения принципа единственной ответственности (Single Responsibility Principle).

Когда у класса много зависимостей, это свидетельствует о том, что он берет на себя больше задач, чем нужно. Теоретически возможно наличие класса со множеством зависимостей, если есть «лишь одна причина, приводящая к изменению класса»[1]. Но чем больше класс, тем меньше вероятность, что он будет придерживаться этого принципа. Использование антипаттерна «Окружающий контекст» скрывает факт излишнего усложнения классов и необходимости реструктуризации приложения.

Использование антипаттерна «Окружающий контекст» из-за своего глобального состояния также усложняет и тестирование. Когда тест меняет глобальное состояние, как показано в коде листинга 5.12, это может повлиять на другие тесты. Такое случается, когда тесты запускаются в параллельном режиме, но могут затрагиваться даже последовательно выполняемые тесты, когда тест «забудет» вернуть назад внесенные им изменения в качестве составной части постпроцессинга и разборки. Несмотря на возможность «смягчения проблем», связанных с тестированием, для этого потребуется специально созданный антипаттерн «Окружающий контекст», а также либо глобальная, либо специально разработанная для теста логика его разборки. Все это связано с усложнениями, от которых избавлен альтернативный вариант.

Использование антипаттерна «Окружающий контекст» затрудняет предоставление разным потребителям разных реализаций зависимости. Предположим, что возникла потребность в том, чтобы часть вашей системы работала с того момента времени, который фиксируется при запуске текущего запроса, в то время как другая ее часть, возможно связанная с выполнением долговременных операций, должна

---

[1] *Мартин Р. К., Ньюкирк Дж. В., Косс Р. С.* Быстрая разработка программ. Принципы, примеры, практика. — М.: Вильямс, 2004. — С. 164.

получить зависимость, обновляемую в реальном времени[1]. В коде, показанном в листинге 5.13, именно это и происходит, то есть потребителям предоставляются разные реализации зависимости:

```
private static readonly ILog Logger =
    LogManager.GetLogger(typeof(MessageGenerator));
```

Чтобы иметь возможность предоставлять потребителям разные реализации, API `GetLogger` требуется, чтобы потребитель передавал соответствующую информацию о типе. Тем самым код потребителя неоправданно усложняется.

Применение антипаттерна «Окружающий контекст» приводит к тому, что использование его зависимости приобретает связанность на временном уровне. Если антипаттерн «Окружающий контекст» не будет проинициализирован в корне композиции, приложение будет давать сбой, только когда класс воспользуется зависимостью в первый раз. Нам бы хотелось, чтобы в таком случае наши приложения давали сбой значительно раньше.

Я использую абстракцию. Что при этом может пойти не так?

Я (Стивен) однажды работал на клиента, у которого была огромная кодовая база, в которой использовалось регистрирование на манер показанного в коде листинга 5.13. Это регистрирование велось в постоянном режиме: поскольку разработчики хотели избавиться от прямой зависимости от проблемной библиотеки регистрирования log4net (https://logging.apache.org/log4net/), они воспользовались другой сторонней библиотекой, предоставляющей им абстракцию в виде надстройки над библиотеками регистрирования. Эта библиотека называлась Common.Logging (https://github.com/net-commons/common-logging). Но помощи не последовало, так как библиотека Common.Logging имитировала API библиотеки log4net, скрывая тот факт, что в их проектах зачастую случайно содержалась зависимость от обеих библиотек. Это привело к тому, что многие классы по-прежнему зависели от библиотеки log4net. Но гораздо важнее то, что, даже при сокрытии разработчиками приложения факта использования log4net за абстракцией, зависимость от сторонней библиотеки по-прежнему сохранилась и теперь это выразилось в том, что каждый класс стал зависеть от использования антипаттерна «Окружающий контекст», предоставленного библиотекой Common.Logging (подобно тому как это произошло в коде листинга 5.13).

Проблема начала проявляться при обнаружении ошибки в Common.Logging, при которой происходил сбой вызова статического метода `GetLogger` на определенных машинах разработчика при запуске внутри IIS. На таких машинах разработчика запуск приложения становился невозможным, поскольку первый вызов `LogManager.GetLogger` давал сбой. К моему несчастью, я как раз и был одним из тех двух разработчиков, столкнувшихся с данной проблемой.

Многие разработчики нашей организации помогали нам в попытках разобраться с произошедшим и потратили несметное количество часов, пытаясь выяснить, что

[1] Возможно, кому-то все это покажется надуманным, но в системах, над которыми мы работали на протяжении многих лет, нам встречалось довольно много ошибок, вызванных запросами, проходящими в полночь или при переходе на летнее время.

происходит, но никто из них не нашел решения или пути обхода возникшей проблемы. В конце концов я закомментировал все вызовы антипаттерна «Окружающий контекст» для тех путей кода, которые мне нужно было запускать локально для моей конкретной функции. На тот момент времени переделка под использование DI была, к сожалению, невозможна.

Не сочтите это за намерение придраться к Common.Logging или к log4net, но, позволяя коду приложения получить зависимость от сторонних библиотек, вы идете на вполне определенный риск. Этот риск возрастает, когда зависимость возникает от используемого в библиотеке антипаттерна «Окружающий контекст».

Мораль сей истории в том, что, используй разработчики не антипаттерн «Окружающий контекст», а правильные паттерны DI, я бы запросто смог локально поменять в корне композиции настроенный регистратор на его имитацию, не требующую загрузки Common.Logging. Несколько минут работы сэкономили бы организации потраченное впустую время.

Хотя использование антипаттерна «Окружающий контекст» не оказывает такого же разрушительного воздействия, как использование антипаттерна «Локатор сервисов», поскольку оно скрывает не произвольное число зависимостей, а всего лишь одну нестабильную зависимость, ему не место в качественно проработанной кодовой базе. Всегда найдутся более удачные варианты, речь о которых пойдет в следующем разделе.

Рефакторинг антипаттерна «Окружающий контекст» и переход к применению технологии DI

Не стоит удивляться, увидев применение антипаттерна «Окружающий контекст» даже в той кодовой базе, разработчики которой прекрасно разбираются в DI и во вреде, наносимом применением антипаттерна «Локатор сервисов». Убедить разработчиков, привыкших к использованию антипаттерна «Окружающий контекст», отказаться от него бывает нелегко. Кроме того, если переделка одного класса под использование DI не представляет особого труда, то избавление от укоренившихся проблем, подобных неэффективным и вредным стратегиям регистрирования, дается намного труднее. Обычно имеется солидный объем кода, причины регистрирования работы которого не всегда ясны. Если прежние разработчики давно покинули организацию, определение возможностей удаления подобных регистрирующих инструкций или их превращения в выдачу исключений может протекать весьма медленно. Тем не менее, если технология DI в кодовой базе уже применяется, переделка применения антипаттерна «Окружающий контекст» в применение DI не вызывает затруднений.

Класс-потребитель антипаттерна «Окружающий контекст» обычно содержит один или несколько его вызовов, возможно разбросанных по нескольким методам. Поскольку первым шагом переделки станет централизация вызова кода антипаттерна «Окружающий контекст», то лучшим местом для нее станет конструктор.

Создайте приватное поле, предназначенное только для чтения и содержащее ссылку на зависимость, и присвойте ему зависимость, содержавшуюся в антипаттерне «Окружающий контекст». Отныне этим новым приватным полем сможет воспользоваться весь остальной код класса. Теперь вызов кода антипаттерна «Окружающий контекст» можно будет заменить параметром конструктора, присваивающим полю значение, и граничным оператором, гарантирующим, что значением параметра конструктора не является `null`. Скорее всего, этот новый параметр конструктора нарушит работу потребителей. Но если технология DI уже была применена, изменения должны будут коснуться лишь корня композиции и тестов для проверки классов. Вполне ожидаемые результаты переделки применительно к `WelcomeMessageGenerator` показаны в листинге 5.14.

Листинг 5.14. Переделка, позволяющая уйти от использования антипаттерна «Окружающий контекст» и перейти к внедрению зависимости через конструктор

```
public class WelcomeMessageGenerator
{
    private readonly ITimeProvider timeProvider;

    public WelcomeMessageGenerator(ITimeProvider timeProvider)
    {
        if (timeProvider == null)
            throw new ArgumentNullException("timeProvider");

        this.timeProvider = timeProvider;
    }

    public string GetWelcomeMessage()
    {
        DateTime now = this.timeProvider.Now;
        ...
    }
}
```

Переделка антипаттерна «Окружающий контекст» относительно проста, поскольку по большей части она будет выполняться в приложении, где уже применена технология DI. Для приложений, где она еще не применяется, прежде чем заниматься переделкой антипаттерна «Окружающий контекст», сначала лучше решить проблемы, связанные с применением антипаттернов «Диктатор» и «Локатор сервисов».

Само понятие «окружающий контекст» выглядит как некий весьма удачный способ доступа к повсеместно используемой сквозной функциональности, но внешность обманчива. Будучи менее проблематичным, по сравнению с антипаттернами «Диктатор» и «Локатор сервисов», «Окружающий контекст» обычно является «прикрытием» для более крупных проблем проектирования, имеющихся в приложении. Паттерны, рассмотренные в главе 4, предоставляют более удачные решения, а в главе 10 будет показан порядок разработки приложений, позволяющий применять во всем приложении регистрирование и другие сквозные функциональности проще и прозрачнее.

Последним антипаттерном, рассматриваемым в этой главе, будет «Ограниченная конструкция». Зачастую результатом появления этого антипаттерна является желание добиться позднего связывания.

5.4. Антипаттерн «Ограниченная конструкция»

Самой большой сложностью правильной реализации DI является перемещение всех классов с зависимостями в корень композиции. Когда вы добьетесь этого, считайте, что вы уже практически прошли весь намеченный путь. Но даже после этого некоторые ловушки, требующие сосредоточенности, все же остаются.

Нередко допускается ошибка, при которой от зависимостей требуется наличие конструктора с определенной сигнатурой. Обычно она возникает из желания добиться позднего связывания, при котором зависимости могут быть определены во внешнем конфигурационном файле, благодаря чему могут подвергаться изменениям без перекомпиляции приложения.

ОПРЕДЕЛЕНИЕ

> Антипаттерн «Ограниченная конструкция» вынуждает все реализации заданной абстракции требовать от их конструкторов наличия одинаковой сигнатуры с целью разрешения позднего связывания.

Следует иметь в виду, что этот раздел относится только к сценариям, требующим позднего связывания. В тех случаях, когда ссылки на все зависимости делаются напрямую из корня приложения, подобная проблема не возникает. Но опять же возможности замены зависимостей без перекомпиляции стартового проекта у вас не будет. Практическое применение антипаттерна «Ограниченная конструкция» показано в листинге 5.15.

Листинг 5.15. Пример антипаттерна «Ограниченная конструкция»

```
public class SqlProductRepository : IProductRepository
{
    public SqlProductRepository(string connectionStr)
    {
    }
}

public class AzureProductRepository : IProductRepository
{
    public AzureProductRepository(string connectionStr)
    {
    }
}
```

Навязывание конкретной сигнатуры конструкторам в реализациях IProductRepository

Все реализации абстракции `IProductRepository` вынуждены иметь конструктор с одной и той же сигнатурой. В данном примере у конструктора должен быть только один аргумент типа `string`. Хотя само наличие у класса зависимости типа `string`