УДК 004.42

И. А. Диомидов

Иркутский государственный университет путей сообщения, г. Иркутск, Российская Федерация

МЕХАНИЗМ ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ КАК МЕТОД ДОСТИЖЕНИЯ СЛАБОГО СЦЕПЛЕНИЯ МЕЖДУ МОДУЛЯМИ

Аннотация. *В работе рассматриваются и освещаются детали такого механизма конструирования программного обеспечения как внедрение зависимостей. Целью статьи является раскрыть тему управления сложностью разработки программного обеспечения. Обозначены пробелы в построении архитектуры объектно-ориентированных программ и обозначены вопросы, естественным образом возникающие при масштабировании программ. Предложен актуальный на сегодня статьи способ обеспечения слабого сцепления между программными модулями. Предложено описание механизма, а также возможности и проблемы его использования.*

Ключевые слова: *программирование, внедрение зависимостей, объектно-ориентированное программирование, паттерны проектирования*

DEPENDENCY INJECTION AS A METHOD OF ACHIEVING LOW COUPLING BETWEEN MODULES

I. A. Diomidov

Irkutsk State Transport University, Irkutsk, the Russian Federation

Abstract: *This article discusses and highlights the details of a software engineering mechanism such as Dependency Injection. The purpose of the article is to reveal the topic of managing the complexity of software development. Gaps in the architecture of object-oriented programs are indicated and issues that naturally arise when scaling programs are indicated. The method of providing low cohesion between software modules, actual at the time of the article, is proposed. A description of the mechanism is proposed, and a description of its possibilities of use and some problems with its use.*

Key words: *programming, dependency injection, object-oriented programming, design patterns*

Введение

Разработка программного обеспечения (ПО) – комплексный процесс. Кроме создания программного продукта, реализующего все функции, требуемых для решения практической задачи, приходится учитывать и косвенные процессы, влияющие на качественные характеристики создаваемого продукта [1-3].

Разрабатываемая программа должна обладать высокой надежностью и корректностью, а для этого должна иметь механизмы обеспечения и проверки этих характеристик. Процесс разработки программы же динамичен и не конечен: продукт после выхода на рынок продолжает дорабатываться. Это приводит к тому, что в нем должна быть предусмотрена возможность расширения, или адаптации к новым требованиям. Со временем, от этого программы сильно масштабируются и не только линейно, но и качественно. Новые требования и условия способны изменить весь подход к разработке.

Это естественным образом порождает необходимость использования определенной единой методологии, подхода к разработке. Цель этой методологии, гарантировать надежность и адаптируемость программы.

Главным техническим императивом разработки программного обеспечения является управление сложностью. Коммерческий провал продукта в области информационных технологий обычно обусловлен тем, что программа с технической точки зрения была трудоемкой для дальнейшей поддержки. Неудачные конструктивные решения влияют на надежность программы, ее расширяемость; и как следствие напрямую влияют на

конкурентоспособность продукта на рынке. Трудоемкость же вызвана высокой уязвимостью кодовой базы к изменениям требований, повышенной сложностью модулей.

Создание подходов и методологий разработки преследует цель получить инструменты управления сложностью. Иными словами, сделать программы простыми для разработчиков, а сама программа должна отвечать определенным характеристикам качества ПО, часть из которых были перечислены выше.

Сцепленность и связность в объектно-ориентированном программировании

Методология программирования тесным образом связана с языковыми средствами. На настоящий момент развитие языковых средств привело к переходу от ассемблерных кодов к высокоуровневым языкам. Среди высокоуровневых языков широкое распространение получила парадигма объектно-ориентированного программирования.

В объектно-ориентированном программировании (ООП) базовой структурной единицей является класс – описание типа сущностей (объектов) с определенной функциональностью и полями, задающими состояние объекта. Однако, сама по себе ООП предлагает скорее вариант организации программы, способ задания структуры, чем способ уменьшить сложность программы. Так с возникновением ООП стали развиваться различные дизайнерские подходы, шаблоны проектирования предлагающие различные варианты компоновки программ и решения практических задач.

В ООП для выявления качества архитектуры программы выделяются меры сцепленности (coupling) и связности (cohesion) [1]. Сцепленность описывает количество и качество связей между модулями, связность отвечает за степень соответствия класса решаемой им задачи. Эти показатели по совместительству являются мерой сложности модулей и связей между ними, а значит и мерой сложности программы. Высокая связность классов при низкой сцепленности описывают максимально возможную простую программы.

Поэтому можно сказать, что разработка инструментов по управлению сцепленностью и связностью модулей – это ключ к управлению сложностью программы, а следовательно и управлением успеха всего продукта в целом.

В ООП языках программирования, поддерживающих эту парадигму, для обеспечения слабой сцепленности имеются такие типы данных как интерфейс и абстрактный класс, реализующих концепцию полиморфизма. Если слабая сцепленность достигается через качественные интерфейсы, то для получения максимальной простоты программы на уровне архитектуры, необходимо задавать связи между объектами строго через интерфейсы.

Однако, задание связей строго через интерфейсы только выглядит как решение. На деле оно порождает ряд качественно иных вопросов. Например, при использовании интерфейсов необходимо решить, в каком месте происходит выбор реализации интерфейса, то есть где происходит создание объекта через оператор new (выделение памяти под объект и вызов конструктора), и какой тип должен быть выбран и какой именно объект за это должен отвечать. Необходимо в результате разработать механизм разрешения этих вопросов.

В самих же языках программирования нет универсального правила того, каким образом задавать связи между этими объектами, их языковые средства предоставляют целый набор вариантов. Можно задавать эту связь через параметры методов, можно задавать через поля, можно задавать через конструктор. Можно создать Singleton с Abstract Fabric, конструирующий объекты нужных интерфейсов, или реализовать Service Locator и получать реализации во время выполнения программы.

К этому стоит добавить, что большинство языков поддерживающих ООП не препятствуют разработке с использованием сильной сцепленности, то есть вообще не используя интерфейсы, задавая связи между классами явно.

Программирование от интерфейсов, на уровне отдельных объектов действительно порождает упрощение. Объект разрабатывается без тесной связи с другими классами, реализуя самостоятельно только некоторый интерфейс взаимодействия, и используя для своей работы другие интерфейсы. Но на уровне архитектуры остается неопределенность, как

минимум в способах задания этих связей и в механизме разрешения зависимостей. Частично эти вопросы позволяют решить шаблоны проектирования. Но только частично, потому что они предлагают специализированное решение задачи, но не универсальный метод построения архитектуры программы в целом. Таким образом, у нас имеются меры сцепленности и связности, но ответственность за контроль сложности программы ложится целиком на программиста, который должен подобрать под задачу подходящий шаблон проектирования.

Внедрение зависимостей

На настоящий момент, наиболее распространенным способом решения описанных выше проблем в ООП является механизм внедрения зависимостей – Dependency Injection (DI). Это комплексный подход в организации кодовой базы, предлагающий стандартизацию построения связей между классами и разрешения зависимостей между ними. Можно сказать, что DI это закономерное развитие концепции программирования от интерфейсов, призванное разрешить возникшую неопределенность на уровне архитектуры программы.

Для реализации этого механизма необходимо использовать следующие архитектурные решения:

- Связи между объектами задаются строго через интерфейс.
- Связь задается внедрение интерфейса в конструкторе зависимого объекта, и эта связь остается неизменной до конца жизни объекта.
- Разрешение зависимостей происходит в Composition Root. Обычно это конкретном приложении, со специальным компонентом, задающим реализацию для того или иного интерфейса.

Таким образом, контроль жизни объекта с реализацией конкретного интерфейса выносится в специализированный участок кода, отвечающий за разрешения зависимостей.

Обычно для обеспечения механизма DI используется фреймворк, предлагающий специальный тип данных – контейнер, способный по описанию (XML файлу, участку кода) автоматически разрешить все необходимые зависимости и создать экземпляры нужных объектов, передав в них реализации зависимостей.

Пакеты программы организуются таким образом:

- Задается доменная логика:
  - интерфейсы, задающие абстрактные внешние типы.
  - доменная модель данных, задающая единые типы данных с информацией для всех зависимых модулей.
  - конкретные классы, реализующие или использующие эти интерфейсы и модель данных.
  - в отдельных модулях создаются конкретные реализации интерфейсов доменной логики.
- Строится приложение с Composition Root, содержащее ссылки на модуль доменной логики и ссылки на модули требуемых для приложения реализаций.

Пример

В качестве примера рассмотрим поэтапно разработку простейшего приложения, получающее информацию о клиентах сайта из базы данных Sql Server, и сохраняющее это по запросу в виде некоторого XML файла. Для реализации этого приложения поступим следующим образом.

1. Задаем доменную модель

1.1. Создадим POCO (Plain Old CLR Object) класс User содержащий информацию о клиентах.

1.2. Создадим интерфейсы IUserRepository (ничего не принимает и возвращает список User), и IReportPrinter (принимает список Users и ничего не возвращает)

1.3. Создадим UserReportService, зависящий от IUserRepository и IReportPrinter, и содержащий один метод MakeReport, ничего не принимающий и ничего не возвращающий.
2. Создаем конкретные реализации IUserRepotitory (SqlUserRepository) и IReportPrinter (XmlReportPrinter). SqlUserRepository будет подключаться к базе данных и выгружать данные. XmlReportPrinter используя XML библиотеку создаст отчет в требуемом формате
3. Создадим приложение, добавив ссылки на пакеты доменной модели и на пакеты реализаций упомянутых выше интерфейсов. В самом приложении опишем Composition Root и укажем чтобы UserReportService использовал реализации SqlUserRepository и XmlReportPrinter описанные на шаге два.

Если необходимо сделать поддержку выгрузки в JSON формате, необходимо будет только создать новый модуль реализации интерфейса IReportPrinter, и изменить в CompositionRoot зависимость с XmlReportPrinter на JsonReportPrinter.

Данный метод обеспечивает механизм управления сложностью на уровне архитектуры. С одной стороны, это стандартизированный метод организации кода. С другой стороны, он позволяет достигнуть всех требуемых характеристик качества ПО.

Для модульного тестирования, достаточно тестировать интерфейсы, заданные в доменной логике. Разделенность доменной логики и конкретных реализаций обеспечивает возможность параллельной разработки - доменная логика никак не связана с реализациями ее интерфейсов, она работает только с интерфейсами. Универсальность и стандартизированность подхода позитивным образом влияет на простоту понимания программы.

Естественно, и этот метод нельзя считать «серебрянной пулей» решающей все проблемы [3]. Например, качество проектирования доменной модели будет все так же влиять на все ее реализации. Разработчику все еще необходимо иметь навыки декомпозиции, качественно проектировать интерфейсы, правильно создавать абстракции. Все еще можно реализовать анти-паттерн Interface soup (каша из интерфейсов), забыв соблюсти принцип Interface segregation (разделения интерфейсов).

Кроме этого, для небольших проектов данный механизм может оказаться довольно громоздким. Описанный выше пример показывает, что для такой примитивной задачи приходится порождать множество абстракций, что не всегда оказывается рациональным. Выгоду от них получается извлечь только в момент, когда нам действительно оказывается приходится менять реализации или использовать сразу несколько, или если нужно вести параллельно разработку нескольких связных модулей.

Для больших проектов, на уровне архитектуры программы механизм DI позволяет добиться управления сложностью программы. Он описывает концептуальное архитектурное решение, как строить архитектуру программы и подходить к разработке. Данный метод позволяет разрабатывать легко масштабируемые приложения, линейно увеличивая сложность с увеличением кодовой базы (модулей и реализуемых возможностей программы), так как общий принцип построения программы остается неизменным. [2]

Заключение

На данный момент, описанный метод получил широкое распространение в языке программирования C# и его Web фреймворке - ASP.NET. Активно используется DI и в чистых Web фреймворках, таких как Angular и ReactJS. Широта распространения, развитие фреймворков свидетельствует об эффективности принципов лежащих в основе механизма внедрения зависимостей.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Макконнелл С. Совершенный Код. Мастер-класс / Пер. с англ. – М.: Издательство «Русская редакция», 2010. – 896 с.

2. Симан М. Внедрение зависимостей в .NET. – СПб.: Питер, 2014. – 464 с.
3. Brooks F.P., jr. No Silver Bullet – Essence and Accident in Software Engineering // Computer Magazine, 1987. – http://www.virtualschool.edu/mon/SoftwareEngineering/ BrooksNoSilverBullet.html.

REFERENCES

1. McConnell S. Code Complete. – Microsoft Press, 2004. – 915 p.
2. Seeman M. Dependency Injection in .NET. – Manning Publication Co, 2012. – 553 p.
3. Brooks F.P., jr. No Silver Bullet – Essence and Accident in Software Engineering // Computer Magazine, 1987. – http://www.virtualschool.edu/mon/SoftwareEngineering/ BrooksNoSilverBullet.html.

Информация об авторах

Диомидов Иван Андреевич - магистрант кафедры «Информационные системы и защита информации», Иркутский государственный университет путей сообщения, г. Иркутск, e-mail: pilad0hwtts@yandex.ru

Information about the authors

Ivan Andreevich Diomidov – master's student of the Department "Information Systems and Information Security", Irkutsk State Transport University, Irkutsk, e-mail: pilad0hwtts@yandex.ru