Принципы проектирования

Пожалуй, важнейший аспект разработки качественного ПО — его проектирование. Создание решений, функционально точных и удобных в сопровождении, — непростая задача, которая в значительной мере зависит от того, какие принципы используются в разработке. Со временем многие решения, принятые на ранних стадиях проекта, могут привести к повышению стоимости и сложности сопровождения, и придется переписывать системы. В то же время продуманную архитектуру можно дополнить и адаптировать к изменениям в технологиях и бизнесе.

Существует множество принципов проектирования и разработки. В этой главе мы отметим несколько самых важных и популярных из них, чтобы вы могли познакомиться с ними поближе.

DRY — «Не повторяйся»

Главный посыл принципа *«Не повторяйся»* (Don't Repeat Yourself, DRY) в том, что дублирование — это просто потеря времени и сил. Дублирование может проявляться в процессах или коде. Множественное выполнение одних тех же требований приводит к выгоранию и хаосу. При первом знакомстве с принципом может быть непонятно, как системе удается дублировать процесс или код. К примеру, однажды кто-то определил порядок выполнения требования, так зачем тратить время на дублирование уже имеющегося функционала? В разработке ПО дублирование происходит часто. Понимание причины случившегося — ключ к осознанию ценности DRY.

Ниже приведены некоторые распространенные причины дублирования кода:

- *недопонимание* — в больших решениях разработчик может не до конца понимать суть и/или не знать, как применять абстрагирование для решения проблемы, используя имеющийся функционал;
- *копирование-вставка* — проще говоря, общая функциональность реализуется с помощью дублирования одного и того же кода вместо проведения рефакторинга во множестве классов.

KISS — «Делай проще, тупица»

Аналогично DRY *«Делай проще, тупица»* (Keep It Simple Stupid, KISS) — важный принцип разработки, существующий уже много лет. KISS делает акцент на том, что главной целью должна быть простота и нужно избегать усложнений. Цель принципа — не допустить ненужных сложностей и уменьшить вероятность ошибок в будущем.

YAGNI — «Вам это не понадобится»

«Вам это не понадобится» (You aren't Gonna Need It, YAGNI) подразумевает, что функционал должен добавляться только при необходимости. Иногда в разработке ПО есть тенденция проектировать с «заделом на будущее», на случай каких-либо изменений. Это может привести к появлению требований, которые не нужны на данный момент или в ближайшем будущем. «Всегда реализуйте функции, которые вам действительно нужны, и никогда, если всего лишь предполагаете, что они понадобятся» (Рон Джеффрис).

MVP — «Продукт с минимальным функционалом»

При использовании подхода *«Продукт с минимальным функционалом»* (Minimum Viable Product, MVP) объем работы ограничивается минимальным набором требований для получения действующего результата. MVP часто комбинируется с другой методологией разработки — Agile (см. раздел «Паттерны жизненного цикла разработки программного обеспечения» далее в этой главе). Количество требований сокращается до какого-то разумного предела, то есть до состояния, когда продукт может быть спроектирован, разработан, протестирован и выпущен. Этот подход уместен при разработке сайтов или приложений, где набор функций может быть введен в эксплуатацию за один цикл разработки.

В главе 3 MVP будет описан на примере гипотетического сценария, в котором этот подход послужит для ограничения охвата изменений, а также для того, чтобы помочь разработчикам соредоточиться на проектировании и составлении требований.

SOLID

SOLID — один из важнейших принципов проектирования, и мы подробно рассмотрим его в главе 3. Фактически составленный из пяти принципов проектирования, SOLID преследует целью создание более удобных в сопровождении и понятных решений. Эти принципы позволяют легче модифицировать код и уменьшают риски возникновения различных проблем.

В главе 3 мы познакомимся с принципами SOLID подробнее, применив их в приложении на C#.

Принцип единственной ответственности

Каждый класс должен нести только одну ответственность. Цель этого принципа — упростить классы и логически их структурировать, так как многозадачные классы слишком сложны для понимания и дальнейшей разработки. Множественная ответственность в данной ситуации — уже причина для изменения. Ответственность

также можно считать одним из аспектов функциональности: «класс... должен иметь одну — и только одну — причину для изменения»[1] (Роберт С. Мартин).

Принцип открытости/закрытости

Данный принцип проектирования лучше всего описать с точки зрения ООП. Средством расширения возможностей класса должно служить наследование. Другими словами, возможность изменения должна быть запланирована и рассмотрена на стадии проектирования класса. Определение и использование интерфейса, реализуемого классом, является применением принципа открытости/закрытости. Класс *открыт* для модификации, в то время как его описание, интерфейс, *закрыто* для нее.

Принцип замещения Лисков

Основной смысл данного принципа состоит в возможности подменять объекты во время работы приложения. В ООП, если класс наследуется от базового класса или реализует интерфейс, на данный класс можно ссылаться как на объект базового класса или интерфейса. Проще будет объяснить это на примере.

Мы определим интерфейс для животного и создадим две его реализации, `Cat` и `Dog`, как показано ниже:

```
interface IAnimal
{
    string MakeNoise();
}
class Dog : IAnimal
{
    public string MakeNoise()
    {
        return "Woof";
    }
}
class Cat : IAnimal
{
    public string MakeNoise()
    {
        return "Meouw";
    }
}
```

Теперь мы можем ссылаться на `Cat` и `Dog`, как на животных:

```
var animals = new List<IAnimal> { new Cat(), new Dog() };
foreach(var animal in animals)
{
    Console.Write(animal.MakeNoise());
}
```

[1] *Мартин Р.* Чистый код: создание, анализ и рефакторинг. — СПб.: Питер, 2021. — С. 167.

Принцип разделения интерфейса

Подобно принципу единственной ответственности, принцип разделения интерфейса гласит: в интерфейсе должны находиться только те методы, которые соблюдают принцип единственной ответственности. Упрощение интерфейса делает код более понятным и облегчает рефакторинг. Ключевая выгода принципа в том, что он помогает изолировать систему от избыточных зависимостей.

Принцип инверсии зависимости

Принцип инверсии зависимости, также известный как принцип внедрения зависимости, гласит: модули должны зависеть от абстракций, а не от деталей реализации. Этот принцип поощряет написание слабо связанного кода, чтобы повысить его читабельность и удобство сопровождения, особенно в крупных и сложных проектах.

Паттерны программного обеспечения

Со временем многие паттерны были сгруппированы в каталоги. В этом разделе в качестве наглядного примера используется два каталога. Первый — ООП-ориентированные паттерны «Банды четырех». Второй относится к интеграции систем и остается технологически независимым. В конце главы будут даны несколько отсылок к дополнительным каталогам и материалам.

Паттерны «Банды четырех»

Возможно, самые значимые и широко известные ООП-коллекции паттернов собраны в книге «Приемы объектно-ориентированного проектирования. Паттерны проектирования» «Банды четырех»[1]. В ней в основном представлены низкоуровневые паттерны, относящиеся к созданию и взаимодействию объектов, а не к более общим архитектурным аспектам. Коллекция состоит из шаблонов, которые можно применять в различных сценариях для создания цельных строительных блоков, в то же время избегая ловушек, распространенных в объектно-ориентированной разработке.

Эрих Гамма, Джон Влиссидес, Ричард Хелм и Ральф Джонсон известны как «Банда четырех» благодаря своим публикациям в 1990-х годах. Книга «Паттерны объектно-ориентированного проектирования» была переведена на несколько языков и содержит примеры на C++ и Smalltalk.

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

Коллекция разбита на три категории: порождающие паттерны, структурные паттерны и поведенческие паттерны, о которых мы и будем говорить ниже.

Порождающие паттерны

С инстанцированием (созданием экземпляров) объектов связаны следующие пять паттернов:

- *«Абстрактная фабрика»* — паттерн для создания объектов, принадлежащих к семейству классов. Конкретный объект определяется во время выполнения;
- *«Строитель»* — полезный паттерн для более сложных объектов, в котором создание экземпляров контролируется вне созданного класса;
- *«Фабричный метод»* — паттерн для создания объектов, наследованных от класса, в котором конкретный класс определяется во время работы;
- *«Прототип»* — паттерн для копирования или клонирования объекта;
- *«Одиночка»* — паттерн, позволяющий создавать только один экземпляр заданного класса.

В главе 3 мы подробно рассмотрим паттерн «Абстрактная фабрика». В главе 4 обсудим паттерны «Фабричный метод» и «Одиночка», включая использование поддерживающего их фреймворка .NET Core.

Структурные паттерны

Со взаимоотношениями между объектами и классами связаны следующие паттерны:

- *«Адаптер»* — паттерн проведения соответствия между двумя разными классами;
- *«Мост»* — паттерн, позволяющий заменять реализацию деталей класса, не прибегая к его модификации;
- *«Компоновщик»* — используется для создания иерархии классов с древовидной структурой;
- *«Декоратор»* — паттерн для замены функционала класса во время выполнения;
- *«Фасад»* — применяется для упрощения сложных систем;
- *«Приспособленец»* — паттерн, используемый для уменьшения расхода ресурсов в сложных моделях;
- *«Заместитель»* — паттерн, используемый для представления другого объекта и обеспечивающий дополнительный уровень контроля между вызывающим и вызываемым объектами.

Паттерн «Декоратор»

Чтобы продемонстрировать структурный паттерн, рассмотрим на примере паттерн «Декоратор». В этом примере будут выдаваться сообщения в консольном приложении. Для начала приведем определение базового сообщения с соответствующим интерфейсом:

```
interface IMessage
{
    void PrintMessage();
}

abstract class Message : IMessage
{
    protected string _text;
    public Message(string text)
    {
        _text = text;
    }
    abstract public void PrintMessage();
}
```

Базовый класс позволяет хранить текстовые фразы и требует, чтобы дочерние классы реализовывали метод `PrintMessage()`. Этот код мы дополним двумя классами.

Первый класс, `SimpleMessage`, выводит текст в консоль:

```
class SimpleMessage : Message
{
    public SimpleMessage(string text) : base(text) { }

    public override void PrintMessage()
    {
        Console.WriteLine(_text);
    }
}
```

Второй класс, `AlertMessage`, тоже выводит текст в консоль, но, помимо этого, выдается короткий звуковой сигнал:

```
class AlertMessage : Message
{
    public AlertMessage(string text) : base(text) { }
    public override void PrintMessage()
    {
        Console.Beep();
        Console.WriteLine(_text);
    }
}
```

Разница между этими классами в том, что в `AlertMessage` предусмотрен звуковой сигнал, а не только вывод текста, как в `SimpleMessage`.

Далее определим базовый класс-декоратор для хранения ссылки на объект `Message`:

```
abstract class MessageDecorator : IMessage
{
    protected Message _message;
    public MessageDecorator(Message message)
    {
        _message = message;
    }

    public abstract void PrintMessage();
}
```

Следующие два класса демонстрируют паттерн «Декоратор», добавляя функционал к нашей уже существующей реализации `Message`.

Первый класс, `NormalDecorator`, выделяет сообщение зеленым цветом:

```
class NormalDecorator : MessageDecorator
{
    public NormalDecorator(Message message) : base(message) { }

    public override void PrintMessage()
    {
        Console.ForegroundColor = ConsoleColor.Green;
        _message.PrintMessage();
        Console.ForegroundColor = ConsoleColor.White;
    }
}
```

Класс `ErrorDecorator` выделяет сообщение красным цветом:

```
class ErrorDecorator : MessageDecorator
{
    public ErrorDecorator(Message message) : base(message) { }

    public override void PrintMessage()
    {
        Console.ForegroundColor = ConsoleColor.Red;
        _message.PrintMessage();
        Console.ForegroundColor = ConsoleColor.White;
    }
}
```

Класс `NormalDecorator` будет выводить текст зеленого цвета, а класс `ErrorDecorator` — красного. Самое важное в данном примере то, что декоратор дополняет поведение объекта `Message`, на который мы ссылаемся.

В завершение примера следующий код показывает, как могут использоваться новые сообщения:

```
static void Main(string[] args)
{
    var messages = new List<IMessage>
    {
        new NormalDecorator(new SimpleMessage("First Message!")),
        new NormalDecorator(new AlertMessage("Second Message with a beep!")),
        new ErrorDecorator(new AlertMessage("Third Message with a beep
                                            and in red!")),
        new SimpleMessage("Not Decorated...")
    };
    foreach (var message in messages)
    {
        message.PrintMessage();
    }
    Console.Read();
}
```

Запуск примера покажет, как разные виды паттерна «Декоратор» можно использовать для изменения заданного функционала (рис. 2.1).

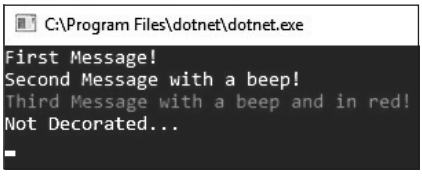

Рис. 2.1

Это упрощенный пример. Но представьте ситуацию, когда у проекта появляется новое требование: знак восклицания должен сопровождаться не коротким звуковым сигналом, а конкретным системным звуком.

```
class AlertMessage : Message
{
    public AlertMessage(string text) : base(text) { }

    public override void PrintMessage()
    {
        System.Media.SystemSounds.Exclamation.Play();
        Console.WriteLine(_text);
    }
}
```

У нас для этого уже есть структура, следовательно, нам достаточно было изменить всего одну строчку, как показано в данном блоке кода.

Поведенческие паттерны

Для определения взаимодействия между классами и объектами могут быть использованы следующие поведенческие паттерны:

- *«Цепочка ответственности»* — паттерн для обработки запросов объектами в коллекции;
- *«Команда»* — паттерн для представления запроса;
- *«Интерпретатор»* — паттерн, определяющий синтаксис или язык для написания инструкций в программе;
- *«Итератор»* — паттерн для обхода коллекции элементов в отсутствие знания подробностей об этих элементах;
- *«Посредник»* — паттерн для упрощения взаимодействия между классами;
- *«Хранитель»* — паттерн для фиксации и сохранения состояния объекта;
- *«Наблюдатель»* — паттерн, позволяющий объектам уведомлять друг друга об изменении своего состояния;
- *«Состояние»* — паттерн для изменения поведения объекта при изменении его состояния;
- *«Стратегия»* — паттерн для реализации коллекции алгоритмов, один из которых может быть применен во время исполнения;
- *«Шаблонный метод»* — паттерн, который определяет этапы алгоритма, делегируя детали реализации подклассу;
- *«Посетитель»* — паттерн, способствующий ослаблению связи между данными и функционалом; позволяет добавлять новые операции, не внося изменений в классы данных.

Паттерн «Цепочка ответственности»

Полезный паттерн, с которым вам следует познакомиться, — это «Цепочка ответственности», поэтому мы и берем его для примера. С его помощью мы создадим коллекцию, или цепочку, классов для обработки запроса. Идея в том, что в ходе обработки запрос должен пройти через каждый класс. Для демонстрации мы возьмем автомобильный сервисный центр, где каждый автомобиль проедет через различные отделы этого предприятия до полного завершения обслуживания.

Начнем с определения коллекции флагов, которые будут использованы для обозначения необходимых сервисов:

```
[Flags]
enum ServiceRequirements
{
```

```
    None = 0,
    WheelAlignment = 1,
    Dirty = 2,
    EngineTune = 4,
    TestDrive = 8
}
```

Атрибут FlagsAttribute в C# предоставляет отличный способ использовать битовое поле для хранения коллекции флагов. Для указания значений перечисления, которые включаются с помощью побитовых операций, будет применяться всего одно поле.

Класс `Car` содержит два поля: одно описывает необходимые услуги, а другое определяет, проведено ли обслуживание:

```
class Car
{
    public ServiceRequirements Requirements { get; set; }

    public bool IsServiceComplete
    {
        get
        {
            return Requirements == ServiceRequirements.None;
        }
    }
}
```

Главное, что можно выделить здесь, — `Car` считает обслуживание завершенным, как только удовлетворяются все требования. Об этом сигнализирует свойство `IsServiceComplete`.

Мы используем базовый абстрактный класс для отображения каждого сервис-техника таким образом:

```
abstract class ServiceHandler
{
    protected ServiceHandler _nextServiceHandler;
    protected ServiceRequirements _servicesProvided;

    public ServiceHandler(ServiceRequirements servicesProvided)
    {
        _servicesProvided = servicesProvided;
    }
}
```

Обратите внимание: услуга, предоставляемая классом, который наследует `ServiceHandler` (то есть сервис-техником), должна быть передана в качестве параметра.

Обслуживание будет выполнено с помощью побитовой операции NOT (~) для *отключения* бита, показывающего необходимость сервиса, в заданном экземпляре `Car` в методе `Service`:

```
public void Service(Car car)
{
    if (_servicesProvided == (car.Requirements & _servicesProvided))
    {
        Console.WriteLine($"{this.GetType().Name} providing
            {this._servicesProvided} services.");
        car.Requirements &= ~_servicesProvided;
    }

    if (car.IsServiceComplete || _nextServiceHandler == null)
        return;
    else
        _nextServiceHandler.Service(car);
}
```

Когда все работы по обслуживанию автомобиля завершены и/или других услуг больше не предусмотрено, цепочка останавливается. Если же остаются другие услуги, а автомобиль еще не готов, то вызывается следующий обработчик.

Данный подход требует установки цепочки, и следующий пример показывает это в действии, используя метод `SetNextServiceHandler()` для запуска в работу очередного сервиса:

```
public void SetNextServiceHandler(ServiceHandler handler)
{
    _nextServiceHandler = handler;
}
```

Сервисные специалисты — это `Detailer`, `Mechanic`, `WheelSpecialist` и инженер `QualityControl`. Класс `ServiceHandler`, представляющий собой `Detailer`, показан в следующем фрагменте кода:

```
class Detailer : ServiceHandler
{
    public Detailer() : base(ServiceRequirements.Dirty) { }
}
```

Механик, в чью специализацию входят настройка и ремонт двигателя, показан в следующем коде:

```
class Mechanic : ServiceHandler
{
    public Mechanic() : base(ServiceRequirements.EngineTune) { }
}
```

Специалист по ходовой части показан в следующем фрагменте кода:

```
class WheelSpecialist : ServiceHandler
{
    public WheelSpecialist() : base(ServiceRequirements.WheelAlignment) { }
}
```

И наконец, контролер качества, который проводит тест-драйв автомобиля:

```
class QualityControl : ServiceHandler
{
    public QualityControl() : base(ServiceRequirements.TestDrive) { }
}
```

Техники сервисного центра определены. Теперь можно обслужить парочку автомобилей. Процесс будет показан в блоке кода `Main`, начиная с создания необходимых объектов:

```
static void Main(string[] args)
{
    var mechanic = new Mechanic();
    var detailer = new Detailer();
    var wheels = new WheelSpecialist();
    var qa = new QualityControl();
```

Следующим шагом будет настройка порядка выполнения разных сервисных работ:

```
    qa.SetNextServiceHandler(detailer);
    wheels.SetNextServiceHandler(qa);
    mechanic.SetNextServiceHandler(wheels);
```

Далее мы сделаем два запроса к механику, который и начнет цепочку ответственности:

```
    Console.WriteLine("Car 1 is dirty");
    mechanic.Service(new Car { Requirements = ServiceRequirements.Dirty });

    Console.WriteLine();

    Console.WriteLine("Car 2 requires full service");
    mechanic.Service(new Car { Requirements = ServiceRequirements.Dirty |
        ServiceRequirements.EngineTune |
        ServiceRequirements.TestDrive |
        ServiceRequirements.WheelAlignment });

    Console.Read();
}
```

Стоит обратить внимание на порядок, в котором оформлена цепочка. На примере сервисного центра сначала механик проводит работы по двигателю, затем за дело

берется специалист по ходовой части. И уже после проводится тест-драйв и машина объявляется прошедшей обслуживание. Изначально тест-драйв был последним этапом, но сервис-центр определил, что, например, в дождливую погоду этот процесс нужно повторить. Может, это немного глупый пример, но он демонстрирует преимущества гибкого определения цепочки ответственности.

На рис. 2.2 показан экран после того, как два автомобиля прошли обслуживание.

Рис. 2.2

Паттерн «Наблюдатель»

Довольно интересен для детального изучения паттерн «Наблюдатель». Он позволяет информировать одни экземпляры о том, что происходит в других. Таким образом, за одним экземпляром может наблюдать множество объектов. На рис. 2.3 показан этот паттерн.

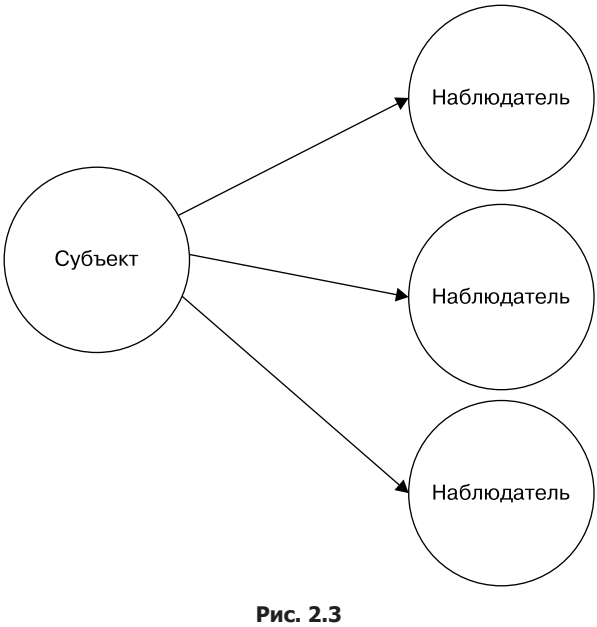

Рис. 2.3

Приведем пример, написав простое консольное приложение на C#, которое создаст один экземпляр класса `Subject` и много экземпляров класса `Observer`. Когда в классе `Subject` меняется значение `quantity`, мы хотим, чтобы каждый экземпляр класса `Observer` получил уведомление об этом.

Класс `Subject` содержит приватное поле `quantity`, которое обновляется публичным методом `UpdateQuantity`:

```
class Subject
{
    private int _quantity = 0;

    public void UpdateQuantity(int value)
    {
        _quantity += value;

        // оповещение наблюдателей
    }
}
```

Для оповещения любых наблюдателей в языке C# предназначены ключевые слова `delegate` и `event`. Ключевое слово `delegate` задает формат или обработчик, который будет вызван. Делегат используется при обновлении `quantity`, как показано в следующем коде:

```
public delegate void QuantityUpdated(int quantity);
```

Делегат определяет `QuantityUpdated` как метод, который получает целочисленное значение и ничего не возвращает. Затем к классу `Subject` добавляется событие:

```
public event QuantityUpdated OnQuantityUpdated;
```

И затем вызывается в методе `UpdateQuantity`, как показано в фрагменте ниже:

```
public void UpdateQuantity(int value)
{
    _quantity += value;

    // оповещение наблюдателей
    OnQuantityUpdated?.Invoke(_quantity);
}
```

В этом примере мы определяем в классе `Observer` метод, который имеет ту же сигнатуру, что и делегат `QuantityUpdated`:

```
class Observer
{
    ConsoleColor _color;
    public Observer(ConsoleColor color)
    {
        _color = color;
    }
```

```
    internal void ObserverQuantity(int quantity)
    {
        Console.ForegroundColor = _color;
        Console.WriteLine($"I observer the new quantity value of {quantity}.");
        Console.ForegroundColor = ConsoleColor.White;
    }
}
```

Данная реализация будет оповещена, когда количество в экземпляре `Subject` изменится, и она выдаст в консоли сообщение соответствующего цвета.

Объединим это все в простом приложении. В начале приложения создаются один объект `Subject` и три объекта `Observer`:

```
var subject = new Subject();
var greenObserver = new Observer(ConsoleColor.Green);
var redObserver = new Observer(ConsoleColor.Red);
var yellowObserver = new Observer(ConsoleColor.Yellow);
```

Далее каждый экземпляр `Observer` подписывается на получение уведомлений об изменении `quantity`, которые рассылает `Subject`.

```
subject.OnQuantityUpdated += greenObserver.ObserverQuantity;
subject.OnQuantityUpdated += redObserver.ObserverQuantity;
subject.OnQuantityUpdated += yellowObserver.ObserverQuantity;
```

А затем мы дважды обновляем `quantity`, как показано ниже:

```
subject.UpdateQuantity(12);
subject.UpdateQuantity(5);
```

Когда приложение запустится, мы получим три разноцветных сообщения, по одному для каждого обновления состояния, как показано на рис. 2.4.

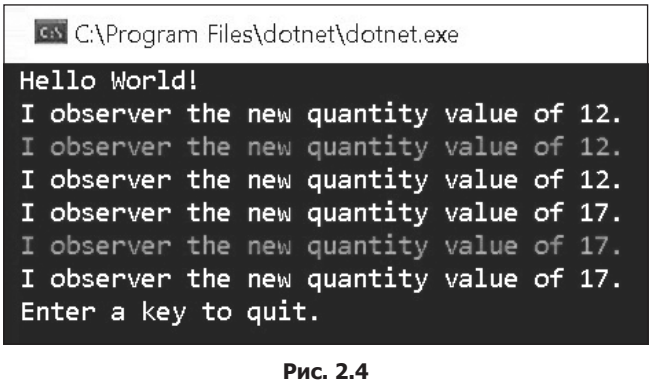

Рис. 2.4

Это был простой пример применения ключевого слова `event` в C#, однако, надеемся, вы теперь понимаете, как можно использовать данный паттерн. Преимуществом

здесь будет уменьшение связи между субъектом и наблюдателями. Субъект не обязан знать о различных наблюдателях.

Паттерны интеграции корпоративных приложений

Интеграция — это область разработки программного обеспечения, преимущества которой основаны на использовании чужих знаний и опыта. В связи с этим существует множество каталогов с паттернами интеграции корпоративных приложений; одни из них не зависят от технологий, в то время как другие рассчитаны на определенный технологический стек. В этом подразделе будут описаны некоторые популярные паттерны интеграции.

В книге «Паттерны интеграции корпоративных приложений»[1] Грегора Хопа и Бобби Вульфа представлена обширная информация о многих паттернах интеграции применительно к различным технологиям. При обсуждении этих паттернов данная книга упоминается чаще всего. Ее оригинал доступен по ссылке www.enterpriseintegrationpatterns.com.

Топология

При интеграции корпоративных приложений важно учитывать топологию систем, между которыми устанавливается связь. В частности, существует две отдельные топологии: веерная система и сервисная шина предприятия.

Топология *«веерная система»* (хаб) описывает паттерн интеграции, где один компонент, хаб, централизован и напрямую сообщается с каждым приложением. Такой подход централизует взаимодействие настолько, что хабу необходимо лишь знать о существовании других приложений, как показано на рис. 2.5.

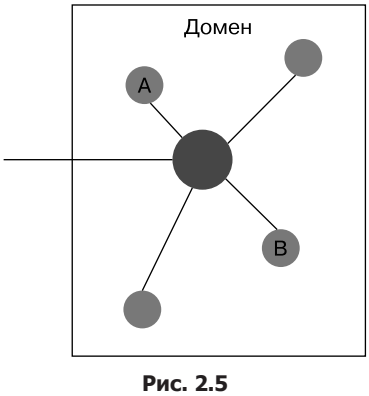

Рис. 2.5

[1] *Хоп Г., Вульф Б.* Шаблоны интеграции корпоративных приложений. — М.: Вильямс, 2016.