5

Основы гигиены

В этой главе будут продемонстрированы предварительные приемы рефакторинга C#. Обычно их можно применять без глубокого понимания предметной области приложения, и в основном они выполняются на уровне синтаксиса. Вы используете их для подготовки к более сложной реструктуризации, где знания предметной области необходимы. Эта глава охватывает следующие основные темы.

- **Мертвый код.** Мертвым кодом называются те разделы кода, которые остались неиспользуемыми после некоторых проведенных модификаций: новое средство реализовано, устранена ошибка или выполнен некоторый рефакторинг. Мертвый код может очень отрицательно влиять на сопровождаемость, и вы узнаете, почему он должен быть исключен.
- **Хорошо инкапсулированный код.** Я напомню о важности хорошо инкапсулированного кода. Инкапсуляция — это основа хорошо сконструированного объектно-ориентированного кода. Вы рассмотрите видимость элемента и уровень доступа как механизмы сокрытия информации и реализации деталей в коде.
- **Явный импорт.** Вы можете ссылаться на элементы из других пространств имен, используя полностью уточненные имена в коде либо применяя директиву `using`. Я сравню оба стиля и объясню, почему предпочитаю применять директиву `using`.
- **Неиспользуемые ссылки на сборки.** Является хорошим тоном удаление таких неиспользуемых ссылок по мере их появления, и вы увидите, как Visual Studio поможет в этом.

В этой главе вы завершите изучение приемов предварительного рефакторинга. Я надеюсь, что по окончании ее чтения вы получите четкую картину чистого кода и гигиены кода, и научитесь выполнять трансформации для приведения в форму любой код, который не соответствует правилам.

Исключение мертвого кода

В первой главе уже упоминалось, что простота — одно из наиболее ценных качеств кода. Каждый дополнительный символ в исходном коде означает дополнительные усилия, необходимые для его понимания. Программисты часто полагают, что код лишним не бывает, и оставляют куски неиспользованного кода внутри работающего

просто на всякий случай. Однако код, который никогда не выполняется, может стать самой большой загадкой для другого программиста, который не писал этот код. Он может подумать, что если код существует, он зачем-то нужен; но поскольку его назначение не очевидно, программист предполагает, что просто еще не достиг нужного понимания кода. Поэтому он тратит больше сил на тестирование, отладку и профилирование, пытаясь понять утерянное значение кода.

Запах: "Мертвый код" (Dead Code)

Обнаружение

Используйте компилятор для поиска неиспользуемого кода. Это можно сделать, закомментировав сомнительный код и пересобрав проект. Если компилятор не выдал никаких ошибок, значит, вы потенциально обнаружили новый случай мертвого кода. Теперь понадобится инспектировать использование во время выполнения этого, возможно, мертвого, кода.

Воспользуйтесь инструментом покрытия тестов (некоторые из них упомянуты в главе 3), чтобы найти потенциально мертвый код. Запустите всесторонний комплект тестов, с активизированным инструментом покрытия. Проанализируйте результаты и найдите код, который не был выполнен. Если код, который не был выполнен — тот же самый код, на отсутствие которого не жалуется компилятор, то довольно вероятно, что вы обнаружили мертвый код. Точность этой методологии в значительной мере зависит от степени охвата кода тестами.

Используйте простой текстовый поиск, чтобы найти закомментированные блоки кода. Закомментированный код точно никогда не выполняется; содержащаяся в нем информация обычно представляет историческую ценность и не должна оставаться частью кодовой базы. Такую информацию лучше хранить в репозитории исходного кода.

Необходимый рефакторинг

Чтобы избавиться от этого запаха, используйте рефакторинг "Исключение мертвого кода" (Eliminate Dead Code).

Обоснование

Мертвый код увеличивает сложность кода. Он затрудняет его понимание и сопровождение; более того, этот код не представляет никакой ценности. Поскольку он никогда не используется, он никогда не выполняется. Мертвый код может привести к излишним усилиям и затуманить начальное предназначение и дизайн результирующего кода.

Наиболее частыми причинами появления мертвого кода являются работы по сопровождению и изменения в дизайне. Возможно, вы удалили строку или две, и не заметили, что где-то целый метод остался неиспользуемым. Или вы добавили некоторое условие во время отладки, чтобы протестировать определенный блок кода, а потом забыли удалить это условие в рабочей версии. Может быть, вы скопировали метод, а потом закомментировали версию метода, который изменили, вместо того, чтобы удалить неиспользуемый метод. Каковой бы ни была причина появления мертвого кода, сохранение его внутри рабочего кода непродуктивно, и такая историческая информация должна быть оставлена только в системе управления версиями.

Определение. *Мертвый код* — это излишний, неработающий код, который не выполняется ни при каких обстоятельствах.

Типы мертвого кода

Различают следующие типы мертвого кода.

- **Недостижимый код.** Это код, о котором вы знаете с абсолютной уверенностью, что он не выполняется никогда. Он по-прежнему учитывается компилятором, который сообщает о любых синтаксических ошибках или тому подобном. Ваша инстинктивная реакция при этом — исправить ошибку, указанную компилятором, но в данном случае это совершенно пустая трата времени и усилий.
- **Закомментированный код.** Этот код игнорируется компилятором, но остается в поле зрения программиста. Если ошибка присутствует в коде, связанном с закомментированным кодом, программист неизбежно начинает искать решение в закомментированном разделе, пытаясь понять, почему код был закомментирован, а не удален. Комментарии могут негативно влиять на визуальное восприятие кода, затрудняя его чтение и понимание.
- **Неиспользуемый код.** Этот код присутствует, если вы имеете дело с некоторого рода повторно используемой библиотекой, например, когда есть метод или класс, о котором вы предполагаете, что он никогда не используется. Вы не можете полностью быть в этом уверенным, потому что не видите полную кодовую базу внутри IDE-среды. В таком случае элемент должен исключаться постепенно. В первой версии известите клиентов, что в будущей версии элемент будет исключен, пометив его атрибутом `ObsoleteAttribute`.

В листинге 5.1 показаны некоторые примеры разновидностей мертвого кода.

Листинг 5.1. Разновидности мертвого кода

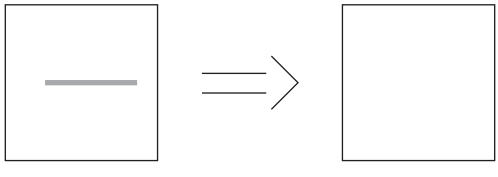
```
using System;
using System.Windows.Forms;
using System.Xml; //System.Xml не имеет ссылок
public class DeadCodeDemo
{
   public static void Main()
   {
      int number = 5;
      //Никогда не оценивается как True, поэтому следующая
      //строка никогда не достигается
      if (number < 4)
      {
         MessageBox.Show("Destined never to show, unreachable");
      }
      //Избыточный код, переменная "number" уже равна 5
      number = 5;
      // Выход из метода, следующая строка никогда не достигается
      return;
      //Следующая строка обнаруживается компилятором
      MessageBox.Show("Destined never to show, unreachable");
   }
   //Метод приватный, но ни разу не используется в классе, к которому принадлежит
   private void NotUsedInClassItBelongsTo()
   {
      MessageBox.Show("Destined never to show, unreachable");
   }
```

```
    // Никогда не используемый общедоступный метод
    public void NotUsed()
    {
       MessageBox.Show("Destined never to show, unused");
    }
    //Закомментированный метод никогда не используется
    //public void Commented()
    //{
    // MessageBox.Show("Destined never to show, commented"
    //}
}
```

Обычные источники мертвого кода

Цикл жизни и смерти — фундаментальный принцип природы. В этом смысле код подобен живому существу. Он рождается, растет и, наконец, умирает. Однажды умерев, он покидает этот мир. Иногда, однако, код остается в этом мире дольше, чем следовало бы. Сейчас вы познакомитесь с некоторыми обычными источниками мертвого кода, которые не удаляются своевременно.

Рефакторинг: "Исключение мертвого кода" (Eliminate Dead Code)



Мотивировка

Больше кода означает большую сложность. Исключая ненужный код из кодовой базы, вы облегчаете его чтение, понимание и сопровождение.

Связанные запахи

Используйте рефакторинг для исключения запаха мертвого кода.

Механизмы

Обратитесь к определению мертвого кода, приведенному ранее в этой главе, чтобы узнать, как обнаруживать мертвый код. После того идентификации такого кода удалите его. Зафиксируйте исправленную версию в системе управления версиями исходного кода, чтобы ваши действия можно было впоследствии отследить и чтобы сохранить резервную копию.

В случае, когда вы работаете с многократно используемым кодом вроде библиотеки или кода компонента и предполагаете, что некоторый элемент не используется, начните с пометки этого элемента атрибутом `ObsoleteAttribute`. Полностью избавьтесь от него в новой версии.

Первоначальный код

```
//Импорт излишний, поскольку метод ToXml закомментирован
using System.Xml;
public class Customer
{
   private string firstName;
```

```
    private string lastName;
    private string sSN;
    public string FirstName
    {
       get { return firstName; }
       set { firstName = value; }
    }
    public string LastName
    {
       get { return lastName; }
       set { lastName = value; }
    }
    public string SSN
    {
       get { return sSN; }
       set { sSN = value; }
    }
    //public XmlDocument ToXml()
    //{
    // XmlDocument doc = new XmlDocument();
    // string xml;
    // xml = "<Customer>" +
    // "<FirstName>" + FirstName + "</FirstName>" +
    // "<LastName>" + LastName + "</LastName>" +
    // "<SSN>" + SSN + "</SSN>" +
    // "</Customer>";
    // doc.LoadXml(xml);
    // return doc;
    //}
}
```

Код после рефакторинга

```
public class Customer
{
    private string firstName;
    private string lastName;
    private string sSN;
    public string FirstName
    {
       get { return firstName; }
       set { firstName = value; }
    }
    public string LastName
    {
       get { return lastName; }
       set { lastName = value; }
    }
    public string SSN
    {
       get { return sSN; }
       set { sSN = value; }
    }
}
```

Существует множество причин появления мертвого кода внутри кодовой базы. В следующих разделах описаны некоторые из наиболее распространенных источников.

Отсоединенный обработчик событий

Событие никогда не возникает, но код его обработчика присутствует. Это случается очень часто, если вы решаете исключить из формы элемент управления, который ранее служил какой-то цели. Весь связанный с ним код обработчика события сохраняется и должен быть удален вручную.

Невидимый элемент управления

Иногда вы помещаете на форму элемент управления, который закрывается другим элементом управления, либо его размер установлен таким, что он стал невидимым. В других случаях он может быть сделан невидимым или неактивным с самого начала, поэтому никогда не используется. Исключайте такие элементы управления и связанный с ними код обработки событий.

Использование разделов импорта неиспользуемых элементов

Мне часто попадаются устаревшие элементы в разделах `using`. Однажды исключив свойство или метод, либо переместив код между классами, очень легко забыть удалить директивы `using`, которые становятся избыточными в результате таких изменений. Это может показаться безвредным, но такой код может иметь далеко идущие последствия. Когда вы выполняете крупномасштабный рефакторинг, очень важно понимать, как в вашем коде работают зависимости. Во время быстрого просмотра кода часто заключения строятся на основе содержимого раздела `using`. Всего несколько лишних операторов `using` могут полностью дезориентировать.

К счастью, Visual Studio помогает справиться с этой задачей. Можете вызвать встроенное в него средство Remove Unused Usings (Удалить неиспользуемые директивы `using`), выбрав пункт меню Edit⇨IntelliSense⇨Organize Usings⇨Remove Unused Usings (Правка⇨IntelliSense⇨Организовать директивы `using`⇨Удалить неиспользуемые директивы `using`), как показано на рис. 5.1. Здесь же эти директивы можно и отсортировать. Обе операции выполняются одним щелчком на пункте меню Remove and Sort (Удалить и отсортировать). Опция Organize Usings также доступна через контекстное меню (вызываемое щелчком правой кнопкой мыши где-нибудь в редакторе кода Visual Studio).

Игнорирование возвращаемого значения процедуры

Это случается, когда возвращаемое значение игнорируется при вызове функции. То есть клиенты на самом деле не заинтересованы в информации, представленной возвращаемым значением. Возвращаемое значение функции должно быть преобразовано в `void`, а код, отвечающий за возврат значения, исключен. Если возвращаемое значение используется лишь иногда, следует еще раз обдумать то, как написана функция.

Игнорирование возвращаемого параметра процедуры

Когда вызывается функция, значение возвращаемого параметра игнорируется. Опять-таки, если это происходит со всеми клиентами функции, то игнорируемый возвращаемый параметр и связанный с ним код должны быть удалены.

Локальная переменная не считывается

Переменной может быть присвоено значение, но она нигде не считывается, т.е. не используется. Фактически, это еще один случай мертвого кода. Удалите такую переменную.

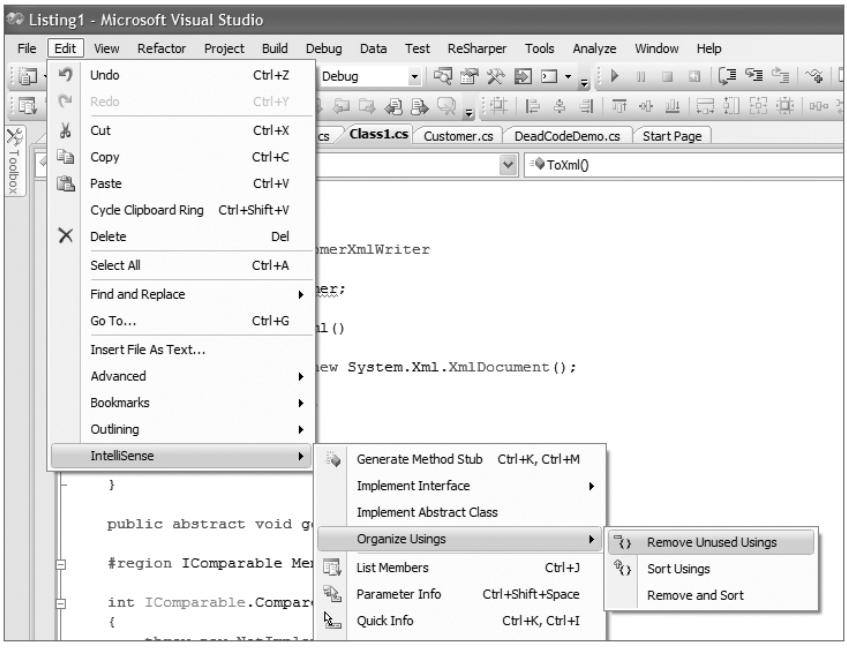


Рис. 5.1. Возможности работы с директивами `using`*, доступные в Visual Studio*

Свойство только для чтения или только для записи включает обе процедуры — `set` и `get`

При написании объявления свойства Visual Studio автоматически создает заготовки процедур `get` и `set` для этого свойства. Очень часто программисты, по инерции или по привычке, оставляют и реализуют обе процедуры, даже если предполагается, что свойство должно быть предназначено только для чтения или только для записи.

Устаревшие элементы

Со временем, по мере внесения все новых и новых изменений, некоторые более крупные элементы кода могут остаться недостижимыми или неиспользуемыми. Такими элементами могут быть классы, перечисления, интерфейсы, модули и даже целые пространства имен.

Не давайте никакой пощады мертвому коду. Используйте какую-нибудь систему управления версиями в качестве средства резервного копирования, и исключайте мертвый код без страха. Если вы не будете этого делать, чаще всего это за вас все равно сделает компилятор. После удаления мертвого кода вы скоро почувствуете себя так, будто сбросили груз прошлого. Ваша кодовая база станет намного яснее и легче для понимания и сопровождения.

Сокращение области видимости и уровня доступа излишне открытых элементов

Вам часто приходилось слышать, что инкапсуляция — первый постулат объектной ориентации. Инкапсуляция применяется для сокрытия внутренних деталей кода от

посторонних глаз. Обычно это называют *сокрытием информации и реализации*. Чтобы снизить сложность систем, вы часто прибегаете к декомпозиции системы на различные модули. Когда вы следуете принципу "разделяй и властвуй", очень важно скрыть как можно больше внутренней информации и деталей реализации каждого индивидуального модуля.

Запах: "Чрезмерная открытость" (Overexposure)

Обнаружение

Если вы подозреваете, что некоторый элемент имеет излишне широкий уровень доступа, сократите его на одну ступень и затем пересоберите проект. Если компилятор не сообщит ни об одной ошибке, значит, вы нашли излишне открытый элемент. Таким элементом может быть интерфейс, модуль, класс, структура, член структуры, процедура, свойство, переменная-член, константа, перечисление, событие, внешнее объявление или делегат.

Аналогично, если вы подозреваете, что определенный элемент имеет слишком широкую область видимости, даже когда специфицирован минимальный уровень доступа, перенесите такой элемент в более ограниченную область и пересоберите проект. Если компилятор не сообщит ни об одной ошибке, значит, вы успешно идентифицировали элемент с избыточно широкой областью видимости.

Необходимый рефакторинг

Используйте рефакторинги "Сокращение уровня доступа" (Reduce Access Level) и "Сокращение контекста" (Reduce Scope), чтобы избавиться от дурного запаха.

Обоснование

Излишняя открытость внутренних деталей реализации противоречит базовому принципу инкапсуляции данных и сокрытия информации. Она делает код менее модульным, более сложным и затрудняет его использование и сопровождение. Зависимости могут свободно расползаться в такой системе, делая ее фактически монолитной. Излишний уровень детализации усложняет использование модулей кода.

В C# самый крупный организационный узел — сборка. Чтобы использовать службы, которые предоставляют некоторые сборки, вам нужен лишь скомпилированный двоичный файл. Вы взаимодействуете со сборкой через интерфейс — набор общедоступно видимых элементов. Обеспечение максимальной простоты этого интерфейса значительно упрощает взаимодействие со сборкой. Вам не нужно обладать никакими знаниями относительно внутренней работы сборки.

Как это работает на практике? Попытаюсь проиллюстрировать на примере. Взгляните на рис. 5.2. Разница в размере между интерфейсами `ShippingCost` и `CartPersistence` наглядно заметна.

Предположим, что имеется система, работающая с тремя сборками: `ShoppingCart`, `ShippingCost` и `CartPersistence`. Вы работаете со сборкой `ShoppingCart` и используете службы сборки `ShippingCost` для вычисления стоимости поставки элементов в корзине покупок. Использование сборки `ShippingCost` достаточно просто: вам, как клиенту, представлен один интерфейс и одна операция.

Теперь вам нужно написать некоторый тестовый код, в котором вы используете также службу `CartPersistence`. Однако вдруг оказывается, что это намного сложнее. Интерфейс `CartPersistence` предоставляет мириады деталей реализации, связанных с коммуникациями с базой данных, управлением транзакциями, протоколированием и т.п., т.е. множеством деталей, о которых ни один клиент, использующий данный ин-

терфейс, вообще не должен беспокоиться. Фактически нет никаких причин, по которым нужно было бы иметь какое-то представление о лежащем в основе механизме постоянного хранения, который может с легкостью использовать простой файл, таблицу Excel или что-то другое. Нет причин, обязывающих клиента `CartPersistence` знать о его внутренней работе. В этом случае, взаимодействуя с `CartPersistence`, вы без всякой необходимости увязаете в болоте его излишне открытых внутренностей.

В качестве эмпирического правила: поддерживайте как можно более ограниченный уровень доступа к программным элементам.

Преимущества инкапсуляции и сокрытия информации и данных легко оценить. Даже визуально сравнивая размеры двух интерфейсов, показанных на рис. 5.2, о них много чего можно сказать. Хорошо инкапсулированные модули легко использовать и легко с ними взаимодействовать, что исключает излишнюю сложность и обеспечивает хорошую модульность системы.

В C# для управления степенью открытости программных элементов можно использовать область видимости и уровень доступа. Минимизируя открытость программных элементов, вы успешно инкапсулируете и скрываете детали реализации кода.

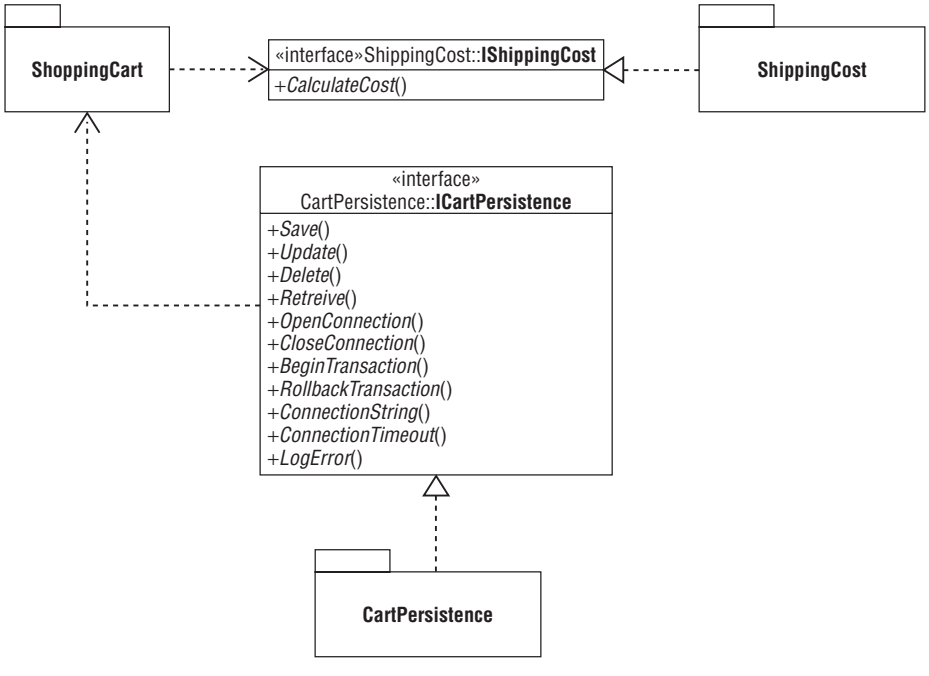


Рис. 5.2. Разница в размере между интерфейсами `ShippingCost` и `CartPersistence`

Область видимости и уровень доступа

Контекст (или область видимости) объявленного элемента определяется областью кода, который имеет доступ к этому элементу без полной его квалификации. Это значит, что для элемента эта область кода находится в ближайшем соседстве, где все знают его по имени. Это имеет также одно очень важное последствие: такие элементы с готовностью отображаются IntelliSense. Элементы вне области видимости должны быть импортированы, чтобы отображаться IntelliSense непосредственно.

Если вы не импортируете элемент, IntelliSense все же поможет найти его, но только если вы укажете полный путь полностью уточненного имени. На рис. 5.3 показано, как IntelliSense помогает сослаться на элемент, находящийся вне текущей области, если вы укажете его полностью уточненное имя.

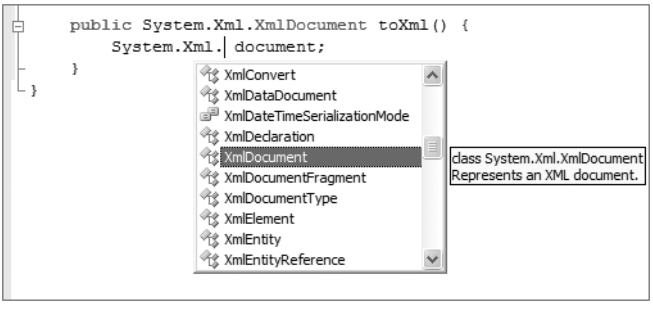


Рис. 5.3. При указании полностью уточненного имени средство IntelliSense помогает сослаться на элемент, находящийся вне текущей области

Область видимости

В C# различают четыре разные области видимости:

- область блока;
- область процедуры;
- область модуля;
- область пространства имен.

Прежде всего, область видимости зависит от закрытого региона, в котором вы объявили элемент — блока, процедуры, класса и структуры. Например, если вы объявляете переменную внутри метода, то она будет доступна в пределах этого же метода, но не доступна никакому другому методу этого или другого класса. В случае если элемент объявлен на уровне модуля (класса или структуры), дальнейшая его область видимости регулируется уровнем доступа. Если переменная объявлена как приватная (`private`), она имеет область видимости внутри модуля, а если используется уровень доступа по умолчанию или общедоступный (`public`), элемент видим в пределах пространства имен (контекст пространства имен).

Уровень доступа

Уровень доступа управляет тем, какой код может обращаться (читать и записывать) объявленный элемент. В C# различаются такие уровни доступа:

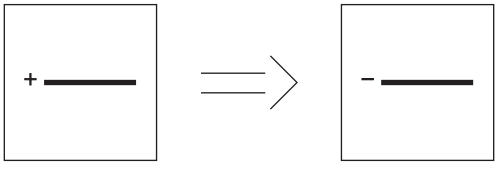
- `public`
- `protected`
- `internal`
- `protected internal`
- `private`

Уровень доступа управляется соответствующими модификаторами доступа при объявлении элемента: `public`, `protected`, `internal` или `private`. В некоторых случаях уровень доступа элемента также зависит от уровня доступа включающей его структуры. Например, вы не сможете обратиться к методу, объявленному как `public`, из другой сборки, если метод, содержащий этот класс, помечен как `internal`.

Обычные источники излишней открытости

Как и в других случаях «дурного запаха», изменения кода — кратчайший путь к избыточной открытости. После некоторого «взбалтывания» кода вы можете оказаться в ситуации, когда область видимости и уровень доступа некоторого элемента должна быть сокращена, потому что теперь обращение к нему идет из более закрытой области. К сожалению, этим шагом часто пренебрегают.

Рефакторинг: "Сокращение уровня доступа" (Reduce Access Level)



Мотивировка

Хорошо инкапсулированный код поддерживает внутренние детали и детали реализации скрытыми от внешнего взгляда. Таким образом, благодаря модульному подходу, сложность системы в целом сокращается. Хорошо инкапсулированный код также помогает контролировать зависимости системы. Другое преимущество сокращения уровня доступа включает снижение вероятности потенциальных конфликтов имен и упрощает управление безопасностью.

Связанные запахи

Используйте рефакторинг для исключения дурного запаха излишней открытости.

Механизмы

Начните с сокращения уровня доступа на один шаг и соберите проект. Если компиляция прошла успешно, повторяйте этот шаг до тех пор, пока не компилятор не выдаст сообщение об ошибке, или не будет достигнут уровень доступа `private`. В случае ошибки вернитесь к наиболее ограниченному уровню доступа, при котором компиляция проходит успешно.

Первоначальный код

```
public class Customer
{
   public string firstName;
   public string FirstName
   {
      get { return firstName; }
      set { firstName = value; }
   }
}
```

Код после рефакторинга

```
public class Customer
{
   private string firstName;
   public string FirstName
   {
      get { return firstName; }
      set { firstName = value; }
   }
}
```

И наоборот, излишняя открытость может быть результатом простого недосмотра. Преимущества инкапсуляции и сокрытия данных и реализации не сразу видны в кратковременной перспективе или в малом масштабе, а потому программисты не уделяют им должного внимания.

Быстрое исправление ошибок дизайна

В некоторых ситуациях быстрое решение некоторых проблем, связанных с дизайном, могут привести к непреднамеренному открытию некоторого приватного элемента. Хотя это дает некоторый быстрый кратковременный результат, огромную цену почти наверняка придется заплатить позднее. Взгляните на код, показанный в листинге 5.2.

Листинг 5.2. Быстрое исправление требует открытия приватного объекта соединения

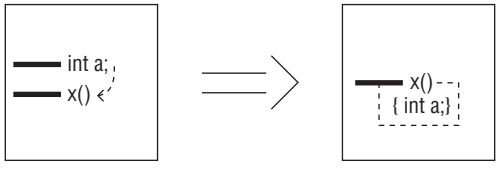
```
using System;
using System.Data.SqlClient;

public class ProductDetailPersistence
{
   private SqlConnection connection;
   //...
   public void Delete()
   {
      //...
   }
   //...
}
```

Класс `ProductDetailPersistence`, как следует из его имени, отвечает за сохранение деталей продукта. В некоторый момент разработчики вдруг замечают очень важное упущение. Не было предусмотрено никакого обеспечения транзакционного поведения класса. Некоторые операции, например, удаление продукта, должны происходить в пределах одной транзакции с удалением деталей этого продукта. Быстрое решение проблемы состоит в открытии объекта соединения объявлением его `public`, тем самым создавая общий транзакционный контекст для обеих операций.

Естественно, это создает проблему, потому что низкоуровневые детали реализации постоянного хранения открываются клиенту. Таким образом, если в некоторый момент будет принято решение изменить лежащее в основе хранилище данных, такое изменение не удастся сохранить в пределах только класса, ответственного за хранение. Поскольку все клиенты знают об `SqlConnection`, любая попытка заменить лежащее в основе хранилище данных механизмом другого типа или другой базой данных, вызовет цепную реакцию изменений в клиентском коде.

Рефакторинг: "Перемещение элемента в более ограниченный регион" (Move Element to a More Enclosing Region)



Мотивировка

Сохранение минимально возможной области видимости элемента помогает программам стать более инкапсулированными, более модульными, более устойчивыми, легкими в сопровождении, чтении и использовании. Другие преимущества — оптимизированное управление памятью, упрощение системы безопасности и т.п.

Связанные запахи

Используйте этот рефакторинг для исключения дурного запаха излишней открытости.

Механизмы

Этот рефакторинг, когда возможно, должен предшествовать рефакторингу "Сокращение уровня доступа" (Reduce Access Level). Выберите элемент для сокращения области видимости и перенесите его объявление в более ограниченный регион.

Первоначальный код

```
using System.Xml;

public class CustomerXmlWriter
{
   private Customer customer;

   //Переменная doc объявлена на уровне класса,
   //но используется только в методе ToXml
   private XmlDocument doc = new XmlDocument();
   //...
   public XmlDocument ToXml()
   {
      string xml;
      xml = "<Customer>" +
            "<FirstName>" + customer.FirstName + "</FirstName>" +
            "<LastName>" + customer.LastName + "</LastName>" +
            "<SSN>" + customer.SSN + "</SSN>" +
            "</Customer>";
      doc.LoadXml(xml);
      return doc;
   }
}
```

Код после рефакторинга

```
using System.Xml;

public class CustomerXmlWriter
{
   private Customer customer;
   //...
```

```
    public XmlDocument ToXml()
    {
        XmlDocument doc = new XmlDocument();
        string xml;
        xml = "<Customer>" +
              "<FirstName>" + customer.FirstName + "</FirstName>" +
              "<LastName>" + customer.LastName + "</LastName>" +
              "<SSN>" + customer.SSN + "</SSN>" +
              "</Customer>";
        doc.LoadXml(xml);
        return doc;
    }
}
```

Как справиться с излишней открытостью

Как только вы идентифицировали излишне открытый элемент, справиться с этой открытостью довольно просто. Первый шаг — снижение уровня доступа. Это ограничение должно внедряться постепенно: необходимо пересобирать проект после каждого сокращения уровня доступа, пока не появится ошибка компилятора или пока не будет достигнут уровень доступа `private`. Если появилась ошибка компиляции, то последнее сокращение уровня доступа должно быть отменено. В табл. 5.1 демонстрируется поэтапный путь сокращения уровня доступа.

Таблица 5.1. Поэтапное сокращение уровня доступа

Текущий	Сокращается до
`public`	`protected` или `internal`
`protected`	`protected internal`
`internal`	`protected internal`
`protected internal`	`private`

Тот же шаблон может быть применен для сокращения области видимости. Этот шаг должен следовать за сокращением уровня доступа. Здесь изменяется местоположение объявления элемента таким образом, что область сокращается до минимально необходимого уровня. В табл. 5.2 показан путь сокращения области видимости.

Таблица 5.2. Поэтапное сокращение области видимости

Текущий	Сокращается до
Вне пространств имен	Пространство имен
Пространство имен	Модуль (класс, структура)
Модуль (класс, структура)	Процедура
Процедура	Блок

Когда идет речь об уровне доступа и области видимости, здесь действительно — чем меньше, тем лучше. При написании кода нужно заботиться о его закрытости.

Выработайте привычку объявлять все элементы в пределах минимальной области видимости и с минимальным уровнем доступа. Это в конечном итоге приведет к лучше спроектированному и лучше инкапсулированному коду.

Использование явного импорта

Проблема явного импорта может поначалу показаться не важной. В конце концов, импорт всего лишь позволяет писать меньше, так что вы можете ссылаться на элемент (пространство имен, интерфейс, модуль, класс, структуру и т.п.), используя его простое имя вместо полностью уточненного имени. Это не избавляет от обязанности добавлять ссылки к проекту, так в чем же важность? Если эффект тот же, почему бы программисту не использовать полностью уточненные имена внутри тела элемента? В следующем разделе я постараюсь ответить на этот вопрос.

Запах: "Использование полностью уточненных имен вне раздела `using`" (Using Fully Qualified Names Outside "Using" Section)

Обнаружение

К сожалению, чтобы обнаружить запах этого рода, придется визуально просмотреть исходный код. Можно, конечно, предпринять шаг, который несколько ускорит поиск. Удалите временно все ссылки проекта. Вдобавок временно закомментируйте раздел `using`, чтобы он не помечался знаком ошибки компилятором. Компилятор теперь пометит все объявления, где используются типы из внешних сборок. Вам останется только просмотреть каждое объявление в поисках полностью уточненных имен в теле кода.

Необходимый рефакторинг

Чтобы избавиться от этого запаха, используйте рефакторинг "Явный импорт" (Explicit Imports).

Обоснование

Использование полностью уточненных имен в теле модуля (класса, структуры, интерфейса и т.п.) затруднит чтение и написание кода. Полностью уточненные имена могут быть очень длинными, и их применение утомительно. Намного лучше импортировать их в одном месте.

Раздел `using` — очень хорошее место для начала исследования крупномасштабного дизайна и зависимостей системы. Несогласованное применение раздела `using` с применением полностью уточненных имен в теле модуля могут создать ложное впечатление при исследовании зависимостей кода.

Раздел `using` отражает зависимости в системе

По мере роста размера проекта и количества людей, принимающих участие в его разработке, такие крупномасштабные вопросы, как зависимости, становятся все более важными и более явными. Естественный способ справиться с крупномасштабными проектами состоит в разделении программного обеспечения на отдельные куски. Таким образом, меньшие группы программистов могут посвятить себя конструированию отдельного элемента мозаики. Поскольку обычно вы начинаете с наброска, демонстрирующего, как эти части стыкуются между собой, конструирование финального продукта сводится к тому, чтобы собрать вместе отдельные куски. В .NET крупнейшим организационными модулями являются сборки.

Рефакторинг: "Замена полностью уточненных имен явным импортом" (Replace Fully Qualified Names with Explicit Imports)

Мотивировка

Добавляя директиву `using`, вы сокращаете объем кода, который должен быть написан. Применение импорта намного эффективнее, чем многократное повторение полностью уточненных имен в коде.

Согласованно применяя раздел `using`, вы предоставляете легко доступную надежную информацию о зависимостях, существующих в коде.

Связанные запахи

Используйте этот рефакторинг для исключения запаха "Использование полностью уточненных имен вне раздела `using`".

Механизмы

1. После идентификации полностью уточненных имен в теле модуля добавьте оператор `using` для этого конкретного имени, если оно уже не присутствует в разделе импорта.
2. Замените найденное полностью уточненное имя в теле модуля простым именем.

Первоначальный код

```
public class CustomerXmlWriter
{
    private Customer customer;
    //...
    public System.Xml.XmlDocument ToXml()
    {
        System.Xml.XmlDocument doc = new System.Xml.XmlDocument();
        string xml;
        xml = "<Customer>" +
              "<FirstName>" + customer.FirstName + "</FirstName>" +
              "<LastName>" + customer.LastName + "</LastName>" +
              "<SSN>" + customer.SSN + "</SSN>" + "</Customer>";
        doc.LoadXml(xml);
        return doc;
    }
}
```

Код после рефакторинга

```
using System.Xml;
public class CustomerXmlWriter
{
    private Customer customer;
    //...
    public XmlDocument ToXml()
    {
        XmlDocument doc = new System.Xml.XmlDocument();
        string xml;
        xml = "<Customer>" +
              "<FirstName>" + customer.FirstName + "</FirstName>" +
              "<LastName>" + customer.LastName + "</LastName>" +
              "<SSN>" + customer.SSN + "</SSN>" + "</Customer>";
        doc.LoadXml(xml);
        return doc;
    }
}
```

Однако если не позаботиться специально, разрабатываемые порознь куски начинают устанавливать непредсказуемые и нежелательные зависимости между собой. Рост новых зависимостей происходит очень просто — добавлением ссылки на чужой проект и одного оператора `using` в код.

Определение. Один модуль (модуль Y) зависит от другого (модуля X), если изменение во втором модуле (X) провоцирует изменения в первом модуле (Y).

Предположим, что вы завершили работу над классом. На следующее утро вы обнаруживаете, что он перестал работать, потому что класс, от которого вы зависите, неожиданно изменился. Другой программист не знал, что ваш класс зависит от его класса, и решил применить изменения, которые счел необходимыми.

Теперь представьте другой сценарий. Ваша сборка имеет дело с предметной областью, но также учитывает хранилище данных. Для этой цели она использует разные сборки для постоянного хранения информации в XML и базе данных. Однако в этом конкретном приложении вы использовали свою сборку из-за ее прикладной функциональности, даже несмотря на то, что никакая функциональность постоянного хранения не нужна. Тем не менее, сборки для работы с хранилищем XML и базой данных все равно должны быть развернуты вместе с вашей сборкой, поскольку она от них зависит.

И это только вершина айсберга, отражающего проблемы, которые связаны с зависимостями. Вы еще увидите в этой книге большинство из них, по мере погружения в более сложные проблемы, связанные с крупномасштабным дизайном. Но даже из этого краткого представления важность корректного управления зависимостями должна быть очевидной.

Пытаясь идентифицировать зависимости в своем коде, программисты имеют естественную склонность визуально сканировать разделы `using` в файлах исходного кода. Однако в некоторых крупномасштабных проектах сканирование всей кодовой базы не только непрактично, но даже может быть и невозможно. Более того, доступная документация и диаграммы часто могут не соответствовать реальному миру — часто они выражают скорее благие намерения, чем реальное состояние дел в данный момент.

Поучительная история

Послушайте анекдот. Мой знакомый должен был выполнить крупномасштабную работу по рефакторингу, чтобы реорганизовать систему и исключить некоторые зависимости из определенных сборок. После того, как он решил, что работа окончена, он триумфально удалил из проекта ссылку на внешнюю сборку и щелкнул на кнопке Build (Собрать). В результате в IDE-среде выскочил список ошибок компилятора, и он обнаружил несколько мест, где имелись ссылки на типы из внешних сборок в виде полностью уточненных имен. Он строил свои предположения на основе разделов импорта. И столкнулся с проблемой из-за вольного применения полностью уточненных имен по всему коду, из-за чего многие места, где имелись ссылки на внешние типы, не были идентифицированы с самого начала.

Хотя есть инструменты, которые могут помочь в решении задачи обнаружения зависимостей в коде, они не всегда доступны. Вот почему поддержание в порядке раздела `using` и отказ от применения полностью уточненных имен в теле модулей очень важно, если вы заинтересованы в том, чтобы код был читабельным и легким в сопровождении.

Удаление неиспользуемых ссылок на сборки

Хотя компилятор C# достаточно умен, чтобы не включать неиспользуемые ссылки в манифест сборки, поддержка неиспользуемых ссылок на сборки в проекте все же имеет серьезные недостатки. Добавление ссылки на внешнюю сборку — важное решение с точки зрения крупномасштабного дизайна, и должно приниматься с большой осторожностью.

Запах: "Неиспользуемые ссылки" (Unused References)

Обнаружение

Вы должны проверять наличие неиспользуемых ссылок после каждого существенного и крупномасштабного рефакторинга. Удалите подозрительные ссылки на сборки и пересоберите проект. Если сборка пройдет без ошибок, значит, вы успешно удалили неиспользуемую ссылку. И наоборот, если ссылка использовалась и не должна быть удалена, компилятор известит об этом следующим сообщением об ошибке:

```
The type or namespace name 'SomeType' could not be found (are you missing a using
directive or an assembly reference?)
```

Не удается найти тип или пространство имен под названием 'НекоторыйТип' (возможно, пропущена директива using или ссылка на сборку?)

Необходимый рефакторинг

Чтобы избавиться от этого запаха, используйте рефакторинг "Удаление неиспользуемых ссылок" (Remove Unused References).

Обоснование

Неиспользуемые ссылки — благодатная почва для излишних зависимостей. Зависимости сильно влияют на дизайн системы с крупномасштабной точки зрения ("с высоты птичьего полета"). Ссылка — это сигнал программистам, что они могут использовать службы, предоставляемые определенной сборкой, тем самым устанавливая излишнюю в противном случае зависимость.

Наличие установленной зависимости в проекте — это своего рода "зеленый свет", который говорит разработчикам, что они могут использовать службы, предоставленные ссылаемой сборкой. В некоторых случаях, особенно после крупномасштабных изменений кода, ссылки устаревают, и зависимости от указанных в них сборок уводят куда-то в другие места. Дизайн изменился, и проект больше не должен зависеть от ссылаемой сборки.

Однако если ссылка не будет удалена из свойств проекта, то высока вероятность что эта сборка в какой-то момент в будущем будет использована вновь. В конце концов, любой разработчик может задать логичный вопрос: "Почему у нас есть ссылка на сборку, если мы не собираемся использовать ее?". Чтобы избежать подобного рода путаницы, избегайте неиспользуемых ссылок и удаляйте их немедленно после появления.

Рефакторинг: "Удаление неиспользуемых ссылок" (Remove Unused References)

Мотивировка

Наличие строгой политики управления ссылками со своевременным удалением неиспользуемых ссылок поможет ограничить зависимости в проекте.

Связанные запахи

Используйте этот рефакторинг для исключения дурного запаха "Неиспользуемые ссылки" (Unused References).

Механизмы

Удостоверьтесь, что сборка не используется в проекте, с применением механизмов, описанных в разделе "Запах: 'Неиспользуемые ссылки' (Unused References)" ранее в главе. Выберите неиспользуемую ссылку в Solution Explorer и затем щелкните на пункте Remove (Удалить) в контекстном меню. Соберите проект, чтобы удостовериться, что ссылка действительно не использовалась.

Базовая гигиена проекта "Прокат автомобилей"

Благодаря юному возрасту приложения "Прокат автомобилей", все разновидности дурных запахов, описанные в этой главе, пока в нем не завелись. Изменения, которые мы провели, ограничились удалением нескольких закомментированных методов и нескольких отсоединенных обработчиков событий. В листинге 5.3 показан один из отсоединенных обработчиков событий, который был исключен после выполнения базового гигиенического рефакторинга приложения.

Листинг 5.3. Обработчик событий, исключенный из приложения "Прокат автомобилей"

```
private void button1_Click(object sender, EventArgs e)
{
    //Тест SqlConnection
    SqlConnection oCn = new SqlConnection(
        "Data Source=TESLATEAM;" +
        "Initial Catalog=RENTAWHEELS;" +
        "User ID=RENTAWHEELS_LOGIN;" +
        "Password=RENTAWHEELS_PASSWORD_123");
    oCn.Open();
}
```

Резюме

В этой главе были продемонстрированы очень общие случаи плохо сопровождаемого, негигиеничного кода. В таких случаях легко заводится код с душком, поэтому нужны эффективные методы для идентификации и устранения его.

Мертвый код часто может стать источником путаницы и дополнительной сложности. Существуют разные пути появления мертвого кода — в виде закомментированных разделов, неиспользуемых обработчиков событий, удаленных элементов управления, игнорируемых параметров и возвращаемых значений методов, и т.д. Какую бы форму он не принимал, весь мертвый код должен быть удален без исключений.

Очень часто программисты дают излишне широкую область видимости определенным программным элементам. Чтобы сконструировать хорошо инкапсулированный код, важно придерживаться принципов сокрытия информации/реализации. Открытие ненужных внутренних деталей противоречит этому принципу, и в таких

случаях можно повысить инкапсуляцию, сократив область видимости и понизив уровень доступа чрезмерно открытых элементов.

Часто при чтении кода вы полагаетесь на раздел импорта, чтобы выявить зависимости, присутствующие в коде. Использование полностью уточненных имен вместо раздела импорта может значительно затруднить понимание зависимостей в коде. Явному импорту следует отдавать предпочтение перед использованием полностью уточненных имен в теле кода. Каждая существующая ссылка однозначно скажет программисту, что ссылаемая библиотека должна применяться из текущей библиотеки. Проблемы возникают, когда устаревшие ссылки не удаляются, потому что такие ссылки позволяют программистам свободно использовать ссылаемые библиотеки. Удаление неиспользуемых ссылок является хорошим тоном.

Следующая глава посвящена стандартным приемам рефакторинга. Эти приемы не могут применяться без хорошего понимания кода и предметной области. Они могут оказать сильное влияние на способ дизайна кода; использовать их — означает иметь дело с основополагающими проблемами, связанными с качеством кода. Я надеюсь, что предварительный рефакторинг, описанный в этой главе, послужит хорошей разминкой перед более сложной работой, которая ждет нас впереди.