6

Проектирование с целью расширяемости

ВАЖНЫЙ АСПЕКТ ПРОЕКТИРОВАНИЯ инфраструктуры состоит в том, чтобы удостовериться, что расширяемость инфраструктуры была тщательно рассмотрена. Это требует понимания стоимости и выгод, связанных с различными механизмами расширяемости. Эта глава помогает решить, какой из механизмов расширяемости — подклассификация, события, виртуальные члены, обратные вызовы и так далее — лучше всего отвечает требованиям вашей инфраструктуры. В этой главе не рассматриваются подробности дизайна этих механизмов. Такие подробности обсуждаются в других частях книги, а эта глава просто содержит перекрестные ссылки на разделы, в которых описаны такие подробности.

Хорошее понимание ООП — необходимая предпосылка к проектированию эффективной инфраструктуры и, в частности, к пониманию понятий, обсуждаемых в этой главе. Однако в этой книге мы не рассматриваем основы ООП, потому что уже есть превосходные книги, полностью посвященные этой теме. Названия некоторых из них приведены в списке рекомендованной литературы в конце книги.

6.1. Механизмы расширяемости

Есть много способов допустить расширяемость в инфраструктурах. Они могут быть выстроены в ряд — от менее сильной, но менее дорогостоящей к очень сильной, но дорогой. Для любого требования расширяемости вы должны выбрать наименее дорогостоящий механизм расширяемости, который отвечает вашим требованиям. Имейте в виду, что обычно можно добавить дополнительную расширяемость позже, но вы никогда не сможете сделать без изменений, не совместимых с предыдущими версиями.

Этот раздел подробно обсуждает некоторые механизмы расширяемости инфраструктуры.

6.1.1. Негерметичные классы

Герметичные (sealed) классы не могут быть унаследованы, и они предотвращают расширяемость. Напротив, классы, которые могут быть унаследованы, называют негерметичными.

```
// Класс String не может быть унаследован
public sealed class String { ... }

// TraceSource может быть унаследован
public class TraceSource { ... }
```

В подклассах можно добавить новые члены, применить атрибуты и реализовать дополнительные интерфейсы. Хотя подклассы могут получить доступ к защищенным членам и заместить виртуальные члены, эти механизмы расширяемости значительно различаются по стоимости и преимуществам. Подклассы описаны в разделах 6.1.2 и 6.1.4. У добавленных к классу защищенных и виртуальных членов могут быть дорогие разветвления, если они сделаны неаккуратно, так что если нужна простая, недорогая, расширяемость, негерметичный класс, в котором не объявлены виртуальные или защищенные члены, является хорошим способом ее реализации.

✓ **РЕКОМЕНДУЕМ** использовать негерметичные классы без добавленных виртуальных или защищенных членов как отличный способ реализации недорогой, но все же очень ценной расширяемости инфраструктуры.

Разработчикам часто нравится наследовать негерметичные классы, чтобы добавить члены для удобства, такие как пользовательские конструкторы, новые методы и перегрузки метода[1]. Например, класс для обмена сообщениями в системе `System.Messaging.MessageQueue` не герметичен и потому позволяет пользователям создавать пользовательские очереди, которые по умолчанию рассматриваются как специфические пути в очереди, и добавить пользовательские методы, которые упрощают API для определенных сценариев (в следующем примере сценарий предназначен для метода постановки объектов типа `Order` в очередь).

```
public class OrdersQueue : MessageQueue {
    public OrdersQueue() : base(OrdersQueue.Path) {
        this.Formatter = new BinaryMessageFormatter();
    }
    public void SendOrder(Order order) {
        Send(order, order.Id);
    }
}
```

[1] Некоторые методы для удобства могут быть добавлены к герметичным типам как методы расширения.

ФИЛ ХААК Поскольку разработки в стиле Test-Driven Development были тепло приняты семейством разработчиков .NET, многие разработчики хотят создавать наследников негерметичных классов (часто динамически используя модель инфраструктуры), чтобы подставить тест-дублер вместо реальной реализации.
По крайней мере, если вы решили (хоть это и неприятно) сделать ваш класс негерметичным, попробуйте сделать главные члены виртуальными, возможно, через шаблон Template Method, чтобы иметь большую степень управляемости.

Классы не герметичны по умолчанию в большинстве языков программирования, и это рекомендуется по умолчанию для большинства классов в инфраструктурах. Расширяемость, предоставленная негерметичными типами, очень ценится пользователями инфраструктуры, и она весьма недорогая, что позволяет обеспечить относительно низкую стоимость испытаний, связанную с негерметичными типами.

ВЭНС МОРРИСОН Ключевое слово в этом совете "РЕКОМЕНДУЕМ". Имейте в виду, что у вас всегда есть возможность разгерметизировать класс в будущем (это изменение не сломает код), но однажды негерметичный класс должен остаться негерметичным навсегда. Кроме того, разгерметизация действительно запрещает некоторую оптимизацию (например, преобразовывая виртуальные вызовы в более эффективные невиртуальные вызовы (и затем встраивание в код)). Наконец, разгерметизация только помогает пользователям управлять созданием класса (иногда это истинно, иногда не совсем). Короче говоря, только очень редко случается, что дизайн оказывается полезно расширяемым в результате случайности. Быть негерметичным является частью контракта класса и его пользователей и, как все остальное в контракте, имеет право быть сознательным и преднамеренным выбором со стороны проектировщика.

6.1.2. Защищенные члены

Защищенные члены сами по себе не обеспечивают расширяемости, но они могут сделать расширяемость более сильной через подклассы. Они могут использоваться, чтобы экспонировать продвинутые опции настройки, излишне не усложняя основной общедоступный интерфейс. Например, свойство `SourceSwitch.Value` защищено, потому что оно предназначено для использования только в расширенных сценариях настройки.

```
public class FlowSwitch : SourceSwitch {
    protected override void OnValueChanged() {
        switch (this.Value) {
            case "None":      Level = FlowSwitchSetting.None; break;
            // случай "Ни один":
            case "Both":      Level = FlowSwitchSetting.Both; break;
            // случай "Оба":
            case "Entering": Level = FlowSwitchSetting.Entering; break;
```

```
            // случай "Вход":
            case "Exiting": Level = FlowSwitchSetting.Exiting; break;
            // случай "Выход":
        }
    }
}
```

Проектировщики инфраструктуры должны быть внимательны при использовании защищенных членов, потому что само название "защищенный" может ввести в заблуждение. Любой пользователь в состоянии создать подкласс негерметичного класса и тем самым получить доступ к защищенным членам, и после этого он сможет применить все действия кодирования, используемые для общедоступных членов, к защищенным членам.

- ✓ **РЕКОМЕНДУЕМ** использовать защищенные члены для расширенной настройки.

  Защищенные члены — отличный способ обеспечить тонкую настройку, не усложняя общедоступный интерфейс.
- ✓ **НАСТОЯТЕЛЬНО РЕКОМЕНДУЕМ** считать защищенные члены в негерметичных классах общедоступными с точки зрения безопасности, документации и анализа совместимости.

  Любой пользователь может наследовать класс и обратиться к защищенным членам.

> **БРЭД АБРАМС** Защищенные члены являются частью вашего публично вызываемого интерфейса, как и общедоступные члены. При проектировании инфраструктуры Framework мы считали защищенные и общедоступные члены примерно эквивалентными. Мы вообще сделали тот же самый уровень обзора и проверки ошибок в защищенных API, какой мы сделали в общедоступных API, потому что их можно вызвать из любого кода, который помещен в подкласс.

6.1.3. События и обратные вызовы

Обратные вызовы — элементы расширяемости, которые позволяют инфраструктуре снова вызывать пользовательский код через делегата. Этих делегатов обычно передают инфраструктуре через параметр метода.

```
List<string> cityNames = ... // Список строк
cityNames.RemoveAll(delegate(string name) {
    return name.StartsWith("Seattle");
});
```

События — особый случай обратных вызовов, который поддерживает удобные и непротиворечивые синтаксические конструкции для того, чтобы предоставить делегата (обработчика событий). Кроме того, средства завершения оператора в Visual Studio и дизайнеры предоставляют справку по использованию основанных на событиях API.

```
var timer = new Timer(1000);
timer.Elapsed += delegate {
    Console.WriteLine("Time is up!");
};
timerStart();
```

Общий дизайн событий обсуждается в разделе 5.4.

Обратные вызовы и события могут использоваться для того, чтобы обеспечить весьма сильную расширяемость, сопоставимую с виртуальными членами. В то же самое время обратные вызовы и даже еще больше события более доступны широкому кругу разработчиков, потому что они не требуют полного понимания объектно-ориентированного проектирования. Кроме того, обратные вызовы могут обеспечить расширяемость во время выполнения, в то время как виртуальные члены могут быть настроены только во время компиляции.

Основной недостаток обратных вызовов: они более тяжеловесны, чем виртуальные члены. Производительность вызова через делегата будет ниже, чем у вызова виртуального члена. Кроме того, делегаты — объекты, и для них нужна память.

Вы должны также учитывать, что, принимая и вызывая делегата, вы выполняете произвольный код в контексте вашей инфраструктуры. Поэтому необходим тщательный анализ всех пунктов расширяемости с помощью обратных вызовов с точки зрения безопасности, правильности и совместимости.

- ✓ **РЕКОМЕНДУЕМ** использовать обратные вызовы, чтобы пользователи могли предоставить пользовательский код, который будет выполняться инфраструктурой.
- ✓ **РЕКОМЕНДУЕМ** использовать события, чтобы пользователи могли настраивать поведение инфраструктуры без понимания объектно-ориентированного проектирования.
- ✓ **НАСТОЯТЕЛЬНО РЕКОМЕНДУЕМ** предпочитать события простым обратным вызовам, потому что они более понятны широкому кругу разработчиков и интегрированы с завершением оператора в Visual Studio.
- ✗ **НЕ РЕКОМЕНДУЕМ** использовать обратные вызовы в чувствительных к производительности API.

КРЖИШТОФ ЦВАЛИНА Вызовы делегата были сделаны намного быстрее в CLR 2.0, но они все еще приблизительно в два раза медленнее, чем прямые вызовы виртуальных членов. Кроме того, основанные на делегатах API, вообще говоря, менее эффективны с точки зрения использования памяти. Сказав это, отметим, что различия являются относительно незначительными и должны иметь значение, только если API вызывается очень часто.

- ✓ **НАСТОЯТЕЛЬНО РЕКОМЕНДУЕМ** использовать новые типы `Func<...>`, `Action<...>` и `Expression<...>` вместо пользовательских делегатов, определяя API с обратными вызовами.

`Func<...>` и `Action<...>` представляют собой универсальные делегаты. Ниже приведено определение их в .NET Framework.

```
public delegate void Action()
public delegate void Action<T1, T2>(T1 arg1, T2 arg2)
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3)
public delegate void Action<T1, T2, T3, T4>(T1 arg1, T2 arg2,
                                            T3 arg3, T4 arg4)
public delegate TResult Func<TResult>()
public delegate TResult Func<T, TResult>(T arg)
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2)
public delegate TResult Func<T1, T2, T3, TResult>(T1 arg1, T2 arg2,
                                                  T3 arg3)
public delegate TResult Func<T1, T2, T3, T4, TResult>(T1 arg1,
                                T2 arg2, T3 arg3, T4 arg4)
```

Они могут использоваться следующим образом.

```
Func<int,int,double> divide = (x,y)=>(double)x/(double)y;
Action<double> write = (d)=>Console.WriteLine(d);
write(divide(2,3));
```

Выражение `Expression<...>` представляет определения функций, которые могут быть откомпилированы и впоследствии вызваны во время выполнения, но могут также быть сериализированы (преобразованы в последовательную форму) и переданы удаленным процессам.

```
Expression<Func<int,int,double>> expression =(x,y)=>(double)x/(double)y;
Func<int,int,double> divide = expression.Compile();
write(divide2(2,3));
```

Обратите внимание: синтаксическая конструкция для создания объекта-выражения `Expression<>` очень похожа на ту, которая используется для создания объекта `Func<>`; фактически, единственное различие состоит в статическом описании типа переменной (выражение `Expression<>` вместо `Func<...>`).

РИКО МАРИАНИ В большинстве случаев, когда нужно действие `Func` или `Action`, все, что нужно выполнить — всего лишь выполнить некоторый код. Тогда нужно выражение `Expression`, если код должен быть проанализирован, преобразован в последовательную форму или оптимизирован прежде, чем он будет выполнен. Выражение `Expression` наводит на мысль о коде, а `Func`/`Action` — о том, чтобы выполнить его.

✓ **НАСТОЯТЕЛЬНО РЕКОМЕНДУЕМ** измерять и анализировать значения производительности, на которую оказывает влияние использование выражения `Expression<...>`, вместо использования делегатов `Func<...>`, и действие `Action<...>`. Типы выражений `Expression<...>` в большинстве случаев логически эквивалентны делегатам `Func<...>` и действиям `Action<...>`. Основное различие между ними состоит в том, что делегаты предназначены для использования в локальных сценариях процесса; а выражения предназначены для случаев, где выгодно и можно вычислить выражение в удаленном процессе или машине.

РИКО МАРИАНИ Удаленность вычисления является видом инцидента. Главное в выражениях `Expression`: они используются тогда, когда необходимо подумать над кодом, который должен быть выполнен, часто над записью выражений, таких как запросы LINQ, причем затем, рассмотрев код в целом и учтя условия выполнения, вы создаете некоторый оптимизированный план выполнения работы. Именно так LINQ к SQL создает единственный фрагмент SQL из композиции почти несвязанных выражений.

Этот план легко может пойти не так, как надо. Вы могли сделать слишком большой анализ выражений или слишком малый. Вы могли израсходовать также много памяти на хранение деревьев выражений или могли избежать всех деревьев, но тогда окажется, что у вас плохая производительность, потому что у вас очень много маленьких анонимных делегатов.

Если вы посмотрите на шаблоны, которые использовались в реализациях LINQ в .NET Framework, то увидите несколько хороших способов использовать следующие конструкции.

- Используйте выражения, только если вы должны "думать" о коде, а не только о том, чтобы выполнить его.
- Вслепую не составляйте и не выполняйте код, который мог быть разумно оптимизирован, если бы вы "подумали" об этом прежде, чем выполнить его.
- Не создавайте системы, которые настолько долго оптимизируют код, что было бы быстрее выполнить этот код непосредственно без оптимизации.
- Оптимизация — не единственное применение деревьев выражений, но очень важное

✓ **НАСТОЯТЕЛЬНО РЕКОМЕНДУЕМ** понять, что, вызывая делегата, вы выполняете произвольный код, и последствия могут быть связаны с безопасностью, правильностью и совместимостью.

БРАЙЕН ПЕПИН Команда, разрабатывавшая формы Windows (Windows Forms team), столкнулась с этой проблемой при написании части кода нижнего уровня в `SystemEvents`. `SystemEvents` определяет статические API, и поэтому данный код должен быть безопасным с точки зрения взаимодействия потоков. Внутри него используются блокировки, чтобы гарантировать безопасность потока. Ранний код в `SystemEvents` захватил бы блокировку и затем вызвал бы событие.

```
lock(someInternalLock) {
    if(eventHandler!=null) eventHandler(sender, EventArgs.Empty);
}
```

Это плохо, потому что вы понятия не имеете, что собирается сделать пользовательский код в обработчике событий. Если пользовательский код сообщает о потоке и ждет на его собственной блокировке, вы, возможно, только что попали в ловушку взаимной блокировки. Лучше закодировать это вот так.

```
EventHandler localHandler = eventHandler;
if(localHandler != null) localHandler(sender, EventArgs.Empty);
```

Теперь код пользователя никогда не будет блокироваться из-за вашей собственной внутренней реализации. Отметьте, что, поскольку присваивание в управляемом коде является атомным (неделимым), блокировка в этом случае вообще не нужна. Но это справедливо не всегда. Например, если бы ваш код должен был проверить больше чем одну переменную, то блокировка была бы нужна.

```
EventHandler localHandler = null;
lock(someInternalLock) {
    if (eventHandler != null && shouldRaiseEvents) {

        localHandler = eventHandler;
    }
}
if(localHandler!=null) localHandler(sender,EventArgs.Empty);
```

ДЖО ДАФФИ В дополнение к блокировке, обратный вызов под блокировкой (как в приведенном примере) может вызвать повторное вхождение. Блокировки на CLR поддерживают рекурсию, так что если обратному вызову так или иначе удастся сделать вызов того же самого объекта, который инициализировал данный обратный вызов, результаты, как правило, не будут хорошими. Блокировки обычно используются для того, чтобы изолировать инварианты, которые временно нарушены, и все же описанный выше механизм может сделать их доступными так, что произойдет повторное вхождение. Само собой разумеется, это может вызвать фантастические исключения и неожиданное поведение.

Сказав это, мы должны отметить, что иногда эта практика необходима. Если бы обратный вызов использовался для принятия решения, как это имело бы место в случае предиката (и это решение должно было бы быть принято под блокировкой), у вас не было бы никакого выбора. Когда такое неизбежно, убедитесь, что тщательно задокументировали ограничения (никаких взаимодействий между потоками, никаких повторных вхождений). И вы должны гарантировать, что если разработчик нарушит эти ограничения, результат не приведет к уязвимости в безопасности. Риск в случае нарушения этого правила обычно больше, чем награда.

6.1.4. Виртуальные члены

Виртуальные члены могут быть замещены для того, чтобы изменить поведение подкласса. Они весьма подобны обратным вызовам в смысле расширяемости, поддерживаемой ими, но лучше в смысле производительности и потребления памяти. Кроме того, виртуальные члены чувствуют себя более естественно в сценариях, которые требуют создания специального вида существующего типа (специализация).

Основной недостаток виртуальных членов: поведение виртуального члена может быть изменено только во время трансляции. А поведение обратного вызова может быть изменено во время выполнения.

Виртуальные члены, как и обратные вызовы (и, возможно, даже больше, чем обратные вызовы), являются дорогостоящими при проектировании, проверке и поддержке, потому что любой вызов виртуального члена может быть замещен непредсказуемыми способами и может выполнить произвольный код. Кроме того, обычно намного больше усилий требуется для того, чтобы ясно определить контракт виртуальных членов, и потому стоимость проектирования и документирования их выше.

КРЖИШТОФ ЦВАЛИНА Часто задают вопрос: следует ли в документации для виртуальных членов указывать, что замещения должны вызвать основную реализацию. Ответ: при замещении нужно сохранить контракт базового класса. Они могут сделать это, вызывая базовую реализацию или некоторыми другими средствами. Довольно редко случается, что единственный способ сохранить (при замещении) контракт члена состоит в том, чтобы вызвать его. В большом количестве случаев вызов базовой реализации — самый простой способ сохранить контракт (и документы должны указать это), но это требуется редко.

Из-за риска и стоимости следует рассмотреть ограничения расширяемости виртуальных членов. Расширяемость с помощью виртуальных членов сегодня должна быть ограничена теми областями, у которых есть ясный сценарий, требующий расширяемости. Этот раздел содержит рекомендации, позволяющие решить, когда позволить такую расширяемость и когда и как ограничить ее.

✕ **НАСТОЯТЕЛЬНО НЕ РЕКОМЕНДУЕМ** делать члены виртуальными, если у вас для этого нет серьезного основания и вы полностью учитываете стоимость, связанную с проектированием, тестированием и поддержкой виртуальных членов.

Изменение виртуальных членов чревато большим риском нарушения совместимости. Кроме того, виртуальные члены медленнее, чем невиртуальные, главным образом, потому что вызовы виртуальных членов не встраиваются в код.

РИКО МАРИАНИ Убедитесь, что вы полностью понимаете свои требования расширяемости, прежде чем принять решение о расширяемости. Частая ошибка: щедро добавлять к классам виртуальные методы и свойства только для того, чтобы затем обнаружить, что необходимая расширяемость все еще не может быть реализована, зато теперь (и навсегда) все работает медленнее.

ЯН ГРЕЙ Опасность: отправляя типы с виртуальными членами, вы гарантируете тонкие и сложные видимые поведения и взаимодействия подкласса. Я думаю, что проектировщики инфраструктуры недооценивают эту опасность. Например, мы нашли, что перечисление элементов `ArrayList` вызывает несколько виртуальных методов для каждого `MoveNext` и `Current`. Устранение проблем производительности могло бы (но, вероятно, это не произошло) сломать определяемые пользователями реализации виртуальных членов в классе `ArrayList`, которые зависят от порядка и частоты вызовов виртуальных методов.

- ✓ **РЕКОМЕНДУЕМ** ограничить расширяемость только тем, что абсолютно необходимо, с помощью шаблона Template Method, описанного в разделе 9.9.
- ✓ **НАСТОЯТЕЛЬНО РЕКОМЕНДУЕМ** предпочитать защищенный доступ общедоступному доступу для виртуальных членов. Общедоступные члены должны поддерживать расширяемость (если требуется), вызывая защищенный виртуальный член.

  Общедоступные члены класса должны обеспечить подходящий набор функциональных возможностей для непосредственных пользователей данного класса. Виртуальные члены проектируются так, чтобы их можно было заместить в подклассах, и защищенный доступ — отличный способ указать те области видимости для всех виртуальных членов, предназначенных для расширения, где они могут использоваться.

```
public Control{
    public void SetBounds(...){
        ...
        SetBoundsCore (...);
    }

    protected virtual void SetBoundsCore(...){
        // Выполните реальную работу здесь.
    }
}
```

Дополнительные сведения по этой теме содержатся в разделе 9.9.

ДЖЕФФРИ РИХТЕР Тип характеризуется тем, что он определяет несколько перегруженных методов для удобства вызывающей программы. Эти методы обычно позволяют вызывающей программе передавать меньше параметров методу, а затем, внутренне, метод вызывает более сложный метод, передавая дополнительные параметры с подходящими значениями по умолчанию. Если тип предлагает такие методы для удобства, эти методы не должны быть виртуальными, но внутренне они должны вызывать один виртуальный метод, который содержит фактическую реализацию данного метода (который может быть перегружен).

6.1.5. Абстракции (абстрактные типы и интерфейсы)

Абстракция — тип, который описывает контракт, но не обеспечивает полную реализацию контракта. Абстракции обычно реализуются как абстрактные классы или интерфейсы, и они идут с довольно полным набором справочной документации, описывающей необходимую семантику типов, реализующих контракт. Некоторые из самых важных абстракций в .NET Framework включают `Stream`, `IEnumerable<T>` и `Object`. В разделе 4.3 обсуждалось, как сделать выбор между интерфейсом и классом при проектировании абстракций.

Расширить инфраструктуры можно путем реализации конкретного типа, который поддерживает контракт абстракции, а использовать этот конкретный тип можно с помощью API инфраструктуры, в которых используется данная абстракция (такие API выполняют действия над данной абстракцией).

Значимую и полезную абстракцию, которая сможет выдержать испытание временем, спроектировать очень трудно. Основная трудность состоит в том, чтобы получить правильный набор членов, не больше и не меньше. Если у абстракции слишком много членов, ее трудно или даже невозможно реализовать. Если у нее слишком мало членов для обещанных функциональных возможностей, она становится бесполезной во многих интересных сценариях. Кроме того, абстракции без первоклассной документации, которая ясно и обстоятельно объясняет все пред- и постусловия, часто заканчивают тем, что в долгосрочной перспективе пользователи отказываются от них. Из-за этого у абстракций очень высокая стоимость разработки.

ДЖЕФФРИ РИХТЕР Интерфейс `ICloneable` — пример очень простой абстракции с контрактом, который явно никогда не документировался. Некоторые типы реализуют метод `Clone` этого интерфейса так, что он выполняет поверхностное копирование объекта, тогда как некоторые реализации выполняют глубинное копирование. Поскольку то, что должен делать метод `Clone` этого интерфейса, полностью никогда не документировалось, при использовании объекта с типом, который реализует `ICloneable`, вы никогда не знаете, что получите. Из-за этого данный интерфейс бесполезен

Слишком много абстракций в инфраструктуре также негативно влияет на удобство и простоту использования инфраструктуры. Часто весьма трудно понять абстракцию, не понимая, как она вписывается в более широкую картину реализаций и API, выполняющих операции над этой абстракцией. Кроме того, названия абстракций и их членов обязательно абстрактны, что часто делает их загадочными и недоступными без предварительного понимания более широкого контекста их использования.

Однако абстракции обеспечивают чрезвычайно сильную расширяемость, которая редко достижима с помощью других механизмов расширяемости. Они лежат в ядре многих архитектурных шаблонов, таких как дополнения к программе, передача управления обратно, конвейеры и т.д. Они также чрезвычайно важны для тестируемости инфраструктур. Хорошие абстракции позволяют избежать тяжелых зависимостей и потому обеспечивают возможность поблочного тестирования. В конце концов, именно абстракциям обязано столь популярное богатство современных объектно-ориентированных инфраструктур.

✗ **НАСТОЯТЕЛЬНО НЕ РЕКОМЕНДУЕМ** предоставлять абстракции, если они не проверены разработкой нескольких конкретных реализаций и API, использующих эти абстракции.

КРЖИШТОФ ЦВАЛИНА Проект PowerCollections является инфраструктурой, расширяющей пространство имен `System.Collections.Generic`. Это был великий источник обратной связи и проверки правильности абстракций, содержавшихся в этом пространстве имен. На основе обратной связи мы устранили несколько проблем дизайна, которые, вероятно, не были бы в противном случае обнаружены до окончания выпуска, после чего уже обычно слишком поздно исправлять абстракции, потому что такие исправления требуют кардинальных изменений.

- ✓ **НАСТОЯТЕЛЬНО РЕКОМЕНДУЕМ** при проектировании абстракции делать тщательный выбор между абстрактным классом и интерфейсом. Подробности по этой теме приведены в разделе 4.3.
- ✓ **РЕКОМЕНДУЕМ** не забывать о тестах в справочной информации для конкретных реализаций абстракций. Такие тесты должны позволить пользователям проверять, правильно ли реализуют контракт их реализации.

ДЖЕФФРИ РИХТЕР Мне нравится то, что сделала группа разработки форм Windows Forms team: они определили интерфейс под названием `System.ComponentModel.IComponent`. Конечно, любой тип может реализовать этот интерфейс. Но группа, которая разработала Windows Forms, также предоставила класс `System.ComponentModel.Component`, который реализует интерфейс `IComponent`. Таким образом, тип мог быть производным от `Component` и тем самым получить реализацию бесплатно или же тип мог быть производным от другого базового класса и затем "вручную" реализовать интерфейс `IComponent`. При наличии доступного интерфейса и базовового класса разработчики получают возможность выбрать тот, который лучше всего подходит для них.

6.2. Базовые классы

Строго говоря, класс становится базовым тогда, когда другой класс окажется его производным. Для целей этого раздела, однако, мы будем пользоваться другим определением: базовый класс — это класс, спроектированный, главным образом, для того, чтобы обеспечить общую абстракцию или чтобы другие классы могли многократно использовать некоторую заданную по умолчанию реализацию посредством наследования. Базовые классы обычно находятся в середине иерархий наследования, между абстракцией в корне иерархии и несколькими пользовательскими реализациями в основании.

Они помогают реализовать абстракции. Например, одна из абстракций инфраструктуры Framework для упорядоченных коллекций элементов — интерфейс `IList<T>`. Реализация `IList<T>` не тривиальна, и поэтому инфраструктура Framework имеет несколько базовых классов, таких как `Collection<T>` и `KeyedCollection<TKey, TItem>`, которые помогают реализовать пользовательские коллекции.

```
public class OrderCollection : Collection<Order>
{
    protected override void SetItem(int index, Order item) {
        if(item==null) throw new ArgumentNullException(...);
        base.SetItem(index,item);
    }
}
```

Базовые классы обычно сами не подходят в качестве абстракций, потому что они имеют тенденцию содержать слишком большой объем реализации. Например,

базовый класс `Collection<T>` содержит большой объем реализации, связанный с тем, что он реализует неуниверсальный интерфейс `IList` (чтобы лучше интегрироваться с неуниверсальными коллекциями) и что эта коллекция элементов хранится в памяти в одном из его полей.

> **КРЖИШТОФ ЦВАЛИНА** Класс `Collection<T>` может также использоваться непосредственно, без необходимости создавать подклассы, но его основное назначение состоит в том, чтобы поддержать простой способ реализации пользовательских коллекций.

Как указывалось ранее, базовые классы могут предоставить неоценимую помощь тем пользователям, которые должны реализовать абстракции, но в то же самое время они могут возлагать и существенную ответственность. Они добавляют некую поверхность, увеличивают глубину иерархий наследования и тем самым концептуально усложняют инфраструктуру. Поэтому базовые классы должны использоваться, только если они имеют существенное значение для пользователей инфраструктуры. Их нужно избегать, если они предоставляют значение только для конструкторов инфраструктуры, — в этом случае нужно строго предпочитать делегирование внутренней реализации, а не наследование от базового класса.

- ✓ **РЕКОМЕНДУЕМ** создавать базовые классы, даже если они не содержат абстрактных членов. Тогда пользователям будет понятно, что класс спроектирован исключительно для того, чтобы быть унаследованным.
- ✓ **РЕКОМЕНДУЕМ** поместить базовые классы в отдельное пространство имен, а не в то, в котором находятся типы для основных сценариев. По определению базовые классы предназначены для продвинутых расширенных сценариев и поэтому не интересны большинству пользователей. Подробности приведены в разделе 2.2.4.
- ✗ **НЕ РЕКОМЕНДУЕМ** добавлять суффикс "Base" к названиям базовых классов, если эти классы предназначены для использования в общедоступных API.

  Например, несмотря на то что класс `Collection<T>` предназначен для создания производных от него, во многих случаях инфраструктуры экспонируют API, типы которых совпадают с базовым классом, а не с его подклассами, главным образом, из-за стоимости, связанной с новым общедоступным типом.

```
public Directory {
    public Collection<string> GetFilenames(){
        return new FilenameCollection(this);
        }

    private class FilenameCollection : Collection<string> {
        ...
    }
}
```

  То, что `Collection<T>` — базовый класс, является несущественным для пользователя метода `GetFilename`, и потому суффикс "Base" только отвлекал бы пользователей метода.

6.3. Герметизация

Одна из особенностей объектно-ориентированных инфраструктур — то, что разработчики могут расширить и настроить их способами, которые проектировщики инфраструктуры предвидеть не могли. Это — и мощность, и опасность расширяемого дизайна. Поэтому, когда вы проектируете свою инфраструктуру, очень важно при проектировании тщательно спланировать расширяемость, когда она желательна, и ограничить расширяемость, когда она опасна.

КРЖИШТОФ ЦВАЛИНА Иногда проектировщики инфраструктуры хотят ограничить расширяемость иерархии типа установленным набором классов. Например, вы хотите создать иерархию живых организмов, которая разбита на две и только две подгруппы: животные и растения. Один способ сделать это состоит в том, чтобы сделать конструктор `LivingOrganism` внутренним и затем обеспечить два подкласса (растения `Plant` и животные `Animal`) в той же самой сборке и дать им защищенные конструкторы. Поскольку конструктор `LivingOrganism` является внутренним, третьи лица могут только расширить класс животных `Animal` и растений `Plant`, но не `LivingOrganism`.

```
public class LivingOrganism {
    internal LivingOrganism(){} // внутренний конструктор LivingOrganism
    ...
}
public class Animal : LivingOrganism {
    protected Animal() {} // защищенный конструктор для класса Animal
    ...
}
public class Plant : LivingOrganism {
    protected Plant() {} // защищенный конструктор для класса Plant
    ...
}
```

Сильный механизм, который предотвращает расширяемость, — герметизация. Вы можете герметизировать весь класс или отдельные его члены. Герметизация класса препятствует тому, чтобы пользователи наследовали этот класс. Герметизация члена препятствует тому, чтобы пользователи замещали данный член.

```
public class NonNullCollection<T> : Collection<T>
{
    protected sealed override void SetItem(int index, T item)
    {
        if (item == null) throw new ArgumentNullException();

        base.SetItem(index, item);
    }
}
```

Поскольку одним из ключей к дифференциации инфраструктур является то, что они предлагают некоторую степень расширяемости, герметичные классы и члены,

вероятно, будут испытывать на себе очень выраженное неприятие разработчиков, использующих инфраструктуру. Поэтому вы должны герметизировать только то, что необходимо.

✗ **НАСТОЯТЕЛЬНО НЕ РЕКОМЕНДУЕМ** герметизировать классы, не имея на то серьезного основания.

Герметизация класса по той причине, что вы не можете придумать расширенные сценарии, не является серьезным основанием. Пользователям инфраструктуры нравится наследовать классы по различным неочевидным причинам, например добавлять члены для удобства. Примеры неочевидных причин, по которым пользователи хотят наследовать члены, приведены в разделе 6.1.1.

Серьезные основания герметизации класса включают следующее.

- Класс является статическим. Дополнительная информация о статических классах содержится в разделе 4.5.
- Класс в унаследованных защищенных членах хранит тайны, важные с точки зрения безопасности.
- Класс унаследовал много виртуальных членов, и стоимость герметизации их по одному перевесила бы выгоды от того, чтобы оставить класс негерметичным.
- Класс представляет собой атрибут, который требует очень быстрого поиска во время выполнения. Уровень производительности у герметичных атрибутов немного выше, чем у негерметичных. Дополнительная информация по разработке атрибутов содержится в разделе 8.2.

БРЭД АБРАМС Наличие классов, открытых для некоторой настройки, является одним из основных отличий инфраструктуры от библиотеки. В случае библиотеки API (такой, как API Win32), вы, в основном, получаете только то, что получаете. Очень трудно расширять структуры данных и API. Клиенты могут расширить инфраструктуру, такую как MFC или AWT, и настроить ее классы. Увеличение производительности от этого очевидно.

КРЖИШТОФ ЦВАЛИНА Разработчики часто спрашивают: какова стоимость герметизации индивидуальных членов. Стоимость является относительно маленькой, но она отлична от нуля и должна быть принята во внимание. Существует стоимость разработки (количество замещений), стоимость проверки или тестирования (базовый класс вызван из замещения?), стоимость размера сборки (новые замещения), а также стоимость рабочего множества (если вызываются и замещения, и основная реализация).

✗ **НАСТОЯТЕЛЬНО НЕ РЕКОМЕНДУЕМ** объявлять защищенные или виртуальные члены в герметичных типах.

По определению герметичные типы не могут быть унаследованы. Это означает, что защищенные члены в герметичных типах нельзя вызвать, а виртуальные методы герметичных типов не могут быть замещены.

✓ **РЕКОМЕНДУЕМ** герметизировать члены, которые вы замещаете.

```
public class FlowSwitch : SourceSwitch
{
    protected sealed override void OnValueChanged() {
        ...
    }
}
```

Проблемы, которые могут появиться из-за введения виртуальных членов (обсуждаемые в разделе 6.1.4), относятся также к замещению, хотя и в немного меньшей степени. Герметизация замещенных элементов ограждает вас от этих проблем, начиная с данного места в иерархии наследования.

Короче говоря, проектирование расширяемости состоит еще и в том, чтобы знать, когда следует ограничить расширяемость, и герметичные типы — один из механизмов, позволяющий сделать это.

РЕЗЮМЕ

Проектирование расширяемости — критический аспект проектирования инфраструктур. Понимание стоимости и преимуществ, предоставленных различными механизмами расширяемости, позволяет разработать инфраструктуры, которые гибки и свободны от многих ловушек, которые впоследствии приводят к неприятностям.