Kapitel 3

Schnittstellen und Entwurfs-Patterns

Am Ende dieses Kapitels werden Sie in der Lage sein, die folgenden Aufgaben durchzuführen:

- Definieren von Schnittstellen und Beschreiben der wichtigsten Unterschiede gegenüber Klassen
- Anwenden von Entwurfs-Patterns wie Adapter- und Strategy-Pattern auf Schnittstellen
- Beschreiben der Vielseitigkeit einer Schnittstelle, die durch Duck-Typing, Mixins und flüssige Schnittstellen ermöglicht wird
- Beschreiben der Grenzen von Schnittstellen und Umgehen dieser Einschränkungen
- Erkennen häufiger Anti-Patterns und Missbräuche von Schnittstellen

Die Schnittstelle (engl. interface) ist ein leistungsfähiges Konstrukt in der Microsoft .NET Framework-Entwicklung. Das Schlüsselwort `interface` sieht zwar ganz unscheinbar aus, es steht aber für ein sehr leistungsfähiges Prinzip. Werden Schnittstellen richtig eingesetzt, versehen sie Ihren Code mit Erweiterungspunkten, die ihn außerordentlich adaptiv machen. Es gibt aber auch Situationen, in denen es nicht sinnvoll ist, Schnittstellen einzusetzen; leider werden sie auch in solchen Fällen oft genutzt.

Dieses Kapitel beschreibt kurz die Unterschiede zwischen Klassen und Schnittstellen und erklärt, wie Sie die beiden am besten kombinieren, um den Clientcode vor Implementierungsänderungen zu schützen und Polymorphie zu ermöglichen.

Außerdem beschäftigt es sich mit der Vielseitigkeit von Schnittstellen und erklärt, welch universelles Werkzeug sie in modernen Softwarelösungen sind. Das umfasst einige leistungsfähige Entwurfs-Patterns, die bei richtiger Anwendung (in Kombination mit anderen Patterns aus diesem Buch) Code erzeugen, der hochflexibel ist und einfach an veränderte Anforderungen angepasst werden kann. Diese Eigenschaften sind in agilen Projekten unverzichtbar.

Schnittstellen sind für sich allein gesehen kein Patentrezept. Eine sinnvolle Anreicherung mit Schnittstellen kann einem Projekt sicherlich weiterhelfen, aber die Schnittstellen müssen auch richtig benutzt werden. Dieses Kapitel stellt einige der häufigsten Fehler bei der Verwendung von Schnittstellen vor.

Was ist eine Schnittstelle?

Eine Schnittstelle definiert das Verhalten, das eine Klasse zur Verfügung stellt, aber nicht, *wie* dieses Verhalten implementiert ist. Schnittstellen sind eigenständige Konstrukte, die nicht zu Klassen gehören. Aber es ist eine Klasse nötig, deren Code die Schnittstelle mit Leben füllt.

Schnittstellen werden einerseits durch ihre Syntax definiert, das heißt bezüglich der Programmiersprache: das Schlüsselwort `interface` und alles, was dadurch impliziert wird. Sie werden aber andererseits durch ihre Funktionen definiert: die Konzepte, die sie repräsentieren und ermöglichen.

Syntax

Schnittstellen werden mit dem Schlüsselwort `interface` definiert. Sie können Eigenschaften, Methoden und Ereignisse enthalten, genau wie Klassen. Allerdings kann kein Element einer Schnittstelle mit einem Zugriffsmodifizierer versehen werden: Die implementierende Klasse muss eine Schnittstelle als `public` implementieren. Listing 3–1 zeigt die Deklaration einer Schnittstelle sowie eine denkbare Implementierung.

```
public interface ISimpleInterface
{
    // Diese Methode muss implementiert werden.
    void ThisMethodRequiresImplementation();

    // Diese String-Eigenschaft muss ebenfalls implementiert werden.
    string ThisStringPropertyNeedsImplementingToo
    {
        get;
        set;
    }

    // Diese Integer-Eigenschaft braucht nur einen Getter.
    int ThisIntegerPropertyOnlyNeedsAGetter
    {
        get;
    }

    // Schnittstellen können auch Ereignisse enthalten.
    event EventHandler<EventArgs> InterfacesCanContainEventsToo;
}
// . . .
public class SimpleInterfaceImplementation : ISimpleInterface
{

    public void ThisMethodRequiresImplementation()
```

```
    {

    }

    public string ThisStringPropertyNeedsImplementingToo
    {
        get;
        set;
    }

    public int ThisIntegerPropertyOnlyNeedsAGetter
    {
        get
        {
             return this.encapsulatedInteger;
        }
        set
        {
            this.encapsulatedInteger = value;
        }
    }

    public event EventHandler<EventArgs> InterfacesCanContainEventsToo = delegate { };

    private int encapsulatedInteger;
}
```

List. 3–1 Deklarieren und Implementieren einer Schnittstelle

Das .NET Framework unterstützt bei Klassen nicht das Konzept der Mehrfachvererbung, aber sie bietet die Möglichkeit, mehrere Schnittstellen in derselben Klasse zu implementieren.

Es gibt keine Beschränkung für die Zahl der Schnittstellen, die eine Klasse implementieren kann, aber Sie müssen natürlich überlegen, wie sinnvoll es ist, sehr viele Schnittstellen in derselben Klasse zu implementieren. Die Klasse in Listing 3–2 implementiert zwei Schnittstellen.

```
public interface IInterfaceOne
{
    void MethodOne();
}
// . . .
public interface IInterfaceTwo
{
    void MethodTwo();
}
// . . .
```

```
public class ImplementingMultipleInterfaces : IInterfaceOne, IInterfaceTwo
{
    public void MethodOne()
    {
    }

    public void MethodTwo()
    {
    }
}
```

List. 3–2 Dieselbe Klasse kann mehrere Schnittstellen implementieren

Es ist nicht nur möglich, mehrere Schnittstellen in derselben Klasse zu implementieren, dieselbe Schnittstelle kann auch mehrmals von unterschiedlichen Klassen implementiert werden.

Mehrfachvererbung

Manche Sprachen unterstützen das Konzept der Vererbung von mehreren Basisklassen. C++ erlaubt beispielsweise ein solches Konstrukt. Die .NET Framework-Sprachen erlauben es allerdings nicht und der Compiler gibt eine Warnung aus, falls Sie versuchen, eine Klasse von zwei oder mehr Klassen abzuleiten (Abbildung 3–1).

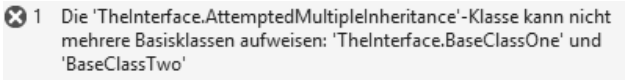

Abb. 3–1 Der Compiler verhindert eine Mehrfachvererbung

Das Diamond-Vererbungsproblem

Das Diamond-Vererbungsproblem (benannt nach der Diamantform, die in Abbildung 3–2 sichtbar wird) ist einer der Gründe, warum Mehrfachvererbung verboten ist. Dieses Problem tritt auf, wenn eine Klasse von mehreren Basisklassen abgeleitet wird. Wenn jede dieser Basisklassen dieselbe Methode enthält, welche soll dann benutzt werden? Abbildung 3–2 zeigt ein UML-Diagramm (Unified Modeling Language) des Problems.

→

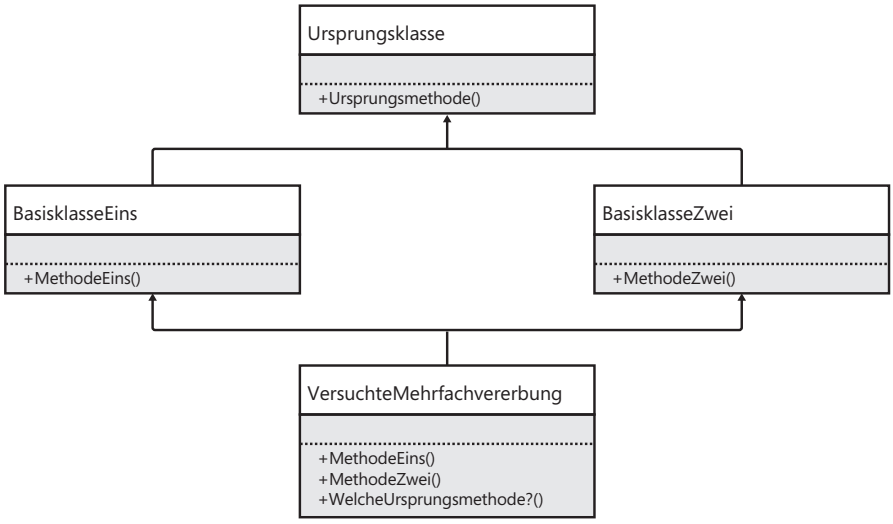

Abb. 3–2 Das Diamond-Vererbungsproblem

Welche Version von `Ursprungsmethode` soll die Klasse `VersuchteMehrfachvererbung` erben? Diejenige, die `BasisklasseEins` von `Ursprungsklasse` geerbt hat, oder diejenige, die `BasisklasseZwei` von `Ursprungsklasse` geerbt hat? Wegen dieser Unklarheit verbietet das .NET Framework die Mehrfachvererbung bei Klassen.

Explizite Implementierung

Schnittstellen können auch *explizit* implementiert werden. Die explizite Schnittstellenimplementierung unterscheidet sich von der impliziten Schnittstellenimplementierung, die in den vorherigen Beispielen demonstriert wurde. Listing 3–3 zeigt eine ähnliche Klasse wie in den letzten Beispielen, aber sie implementiert ihre Schnittstelle explizit.

```
public class ExplicitInterfaceImplementation : ISimpleInterface
{
    public ExplicitInterfaceImplementation()
    {
        this.encapsulatedInteger = 4;
    }

    void ISimpleInterface.ThisMethodRequiresImplementation()
    {
        encapsulatedEvent(this, EventArgs.Empty);
    }

    string ISimpleInterface.ThisStringPropertyNeedsImplementingToo
    {
```

```
        get;
        set;
    }

    int ISimpleInterface.ThisIntegerPropertyOnlyNeedsAGetter
    {
        get
        {
            return encapsulatedInteger;
        }
    }

    event EventHandler<EventArgs> ISimpleInterface.InterfacesCanContainEventsToo
    {
        add { encapsulatedEvent += value; }
        remove { encapsulatedEvent -= value; }
    }

    private int encapsulatedInteger;
    private event EventHandler<EventArgs> encapsulatedEvent;
}
```

List. 3–3 Explizite Implementierung einer Schnittstelle

Um eine explizit implementierte Schnittstelle nutzen zu können, brauchen Clients einen Verweis auf eine Instanz der Schnittstelle. Ein Verweis auf eine Implementierung der Schnittstelle genügt nicht. Listing 3–4 demonstriert diesen Punkt genauer.

```
public class ExplicitInterfaceClient
{
    public ExplicitInterfaceClient(ExplicitInterfaceImplementation
        implementationReference, ISimpleInterface interfaceReference)
    {
        // Wenn Sie die Kommentarzeichen der folgenden Zeilen löschen, erhalten
        //  Sie Fehler beim Kompilieren.
        //var instancePropertyValue =
        //implementationReference.ThisIntegerPropertyOnlyNeedsAGetter;
        //implementationReference.ThisMethodRequiresImplementation();
        //implementationReference.ThisStringPropertyNeedsImplementingToo = "Hello";
        //implementationReference.InterfacesCanContainEventsToo += EventHandler;

        var interfacePropertyValue =
            interfaceReference.ThisIntegerPropertyOnlyNeedsAGetter;
        interfaceReference.ThisMethodRequiresImplementation();
        interfaceReference.ThisStringPropertyNeedsImplementingToo = "Hello";
        interfaceReference.InterfacesCanContainEventsToo += EventHandler;
    }
```

```
    void EventHandler(object sender, EventArgs e)
    {

    }
}
```

List. 3–4 Ist eine Schnittstelle explizit implementiert, sind die Schnittstellenmethoden in Klasseninstanzen nicht sichtbar

Die explizite Implementierung ist eigentlich nur dann nützlich, wenn Sie einen Signaturkonflikt vermeiden wollen, weil die Klasse bereits eine Methode mit derselben Signatur besitzt, wie sie für die Schnittstelle implementiert werden muss.

Jede Methode, die im .NET Framework definiert werden kann, hat eine eindeutige Methodensignatur. Diese Signatur hilft dabei, Methoden eindeutig zu identifizieren und Methoden zu unterscheiden, die überschrieben wurden. Die Signatur einer Methode besteht aus ihrem Namen und ihrer Parameterliste. Auch die Zugriffsebene einer Methode sowie `abstract`- oder `sealed`-Modifizierer fließen in die Methodensignatur ein. Listing 3–5 zeigt eine Vielzahl von Methodensignaturen, von denen einige einen Konflikt auslösen. Ein solcher Konflikt entsteht, wenn Methodensignaturen in allen aufgezählten Aspekten identisch sind. Kein `class`-, `interface`- oder `struct`-Element darf Methoden enthalten, deren Signaturen einen Konflikt auslösen.

```
public class ClashingMethodSignatures
{
    public void MethodA()
    {

    }

    // Konflikt mit der vorherigen Methode.
    //public void MethodA()
    //{

    //}

    // Ebenfalls ein Konflikt: Rückgabewerte fließen nicht in die Signatur ein.
    //public int MethodA()
    //{
    //    return 0;
    //}

    public int MethodB(int x)
    {
        return x;
    }

    // Diesmal kein Konflikt: Die Parameter unterscheiden sich.
```

```
    // Dies ist eine Überladung der vorherigen MethodB.
    public int MethodB(int x, int y)
    {
        return x + y;
    }
}
```

List. 3–5 Einige der Methoden haben dieselbe Signatur

Weil Eigenschaften keine Parameterlisten haben, werden sie nur anhand ihres Namens unterschieden. Die Signaturen von zwei Eigenschaften lösen daher einen Konflikt aus, wenn sie denselben Namen tragen.

Nehmen wir an, die Klasse aus Listing 3–6 soll die Beispielschnittstelle `IInterfaceOne` implementieren.

```
public class ClassWithMethodSignatureClash
{
    public void MethodOne()
    {
    }
}
```

List. 3–6 Wenn diese Klasse die Schnittstelle `IInterfaceOne` implementiert, ergibt sich ein Konflikt bei der Methodensignatur

Weil die Methodensignaturen gleich sind, brauchen Sie lediglich die Schnittstelle zur Klassendeklaration hinzuzufügen (Listing 3–7).

```
public class ClassWithMethodSignatureClash : IInterfaceOne
{
    public void MethodOne()
    {
    }
}
```

List. 3–7 Wenn Sie die Schnittstelle implizit implementieren, können Sie die vorhandenen Methoden dafür nutzen

Wenn ein Client die Schnittstellenmethoden dieser Klasse aufruft, werden die Methoden benutzt, die bereits vorhanden sind. Das ist beispielsweise nützlich, wenn Sie das MVP-Pattern (Model-View-Presenter) in Windows Forms implementieren und zu einer von `Form` abgeleiteten Klasse die Schnittstelle `IView` hinzufügen, für die die Methode `Close` implementiert werden muss. Listing 3–8 zeigt, wie das in der Praxis aussieht.

```
public interface IView
{
    void Close();
}
// . . .
public partial class Form1 : Form, IView
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

List. 3–8 Manchmal ist es ganz praktisch, wenn ein Konflikt bei Methodensignaturen besteht

Muss die Klasse für die Schnittstellenimplementierung dagegen anders implementierte Methoden bereitstellen, sollte sie die Schnittstelle explizit implementieren. Auf diese Weise wird ein Konflikt der Methodensignatur vermieden. Listing 3–9 zeigt eine Klasse, die Methoden, deren Signaturen übereinstimmen, unterschiedlich implementiert.

```
public class ClassAvoidingMethodSignatureClash : IInterfaceOne
{
    public void MethodOne()
    {
        // Ursprüngliche Implementierung
    }

    void IInterfaceOne.MethodOne()
    {
        // Neue Implementierung
    }
}
```

List. 3–9 Wenn Sie eine Schnittstelle explizit implementieren, können Sie Konflikte bei den Methodensignaturen vermeiden

Wenn eine Klasse zwei unterschiedliche Schnittstellen implementieren muss, die nichts miteinander zu tun haben, aber beide eine Methode mit derselben Signatur enthalten, können Sie beide Schnittstellen implizit implementieren. In diesem Fall verwenden beide Schnittstellen dieselbe Methodenimplementierung. Stattdessen können Sie auch beide explizit implementieren, um deutlich zu machen, welche Implementierung zu welcher Schnittstelle gehört (Listing 3–10).

```
public class ClassImplementingClashingInterfaces : IInterfaceOne, IAnotherInterfaceOne
{
    void IInterfaceOne.MethodOne()
    {

    }

    void IAnotherInterfaceOne.MethodOne()
    {

    }
}
```

List. 3–10 Werden zwei Schnittstellen implementiert, die eine identische Methodensignatur enthalten, ist eine explizite Implementierung verständlicher

Polymorphie

Die Fähigkeit, das Objekt eines Typs zu verwenden, während es sich implizit verhält, als hätte es einen anderen Typ, wird als *Polymorphie* (engl. polymorphism) bezeichnet. Clientcode kann auf ein Objekt zugreifen, als hätte es einen bestimmten Typ, obwohl es in Wirklichkeit einen anderen Typ hat. Dieser Programmiertrick ist eines der wichtigsten Konzepte der objektorientierten Programmierung (Object-Oriented Programming, OOP). Es ist die Grundlage, auf der einige der elegantesten adaptiven Lösungen für Programmierprobleme aufbauen.

Abbildung 3–3 zeigt die Schnittstelle `IVehicle`, die das Verhalten von Fahrzeugen definiert, und einige mögliche Implementierungsklassen für Autos (`Car`), Motorräder (`Motorcycle`) und Motorboote (`Speedboat`). Offensichtlich unterscheiden sich diese drei Fahrzeugtypen recht stark, aber dank der einheitlichen Schnittstelle zeigen sie alle dasselbe Verhalten.

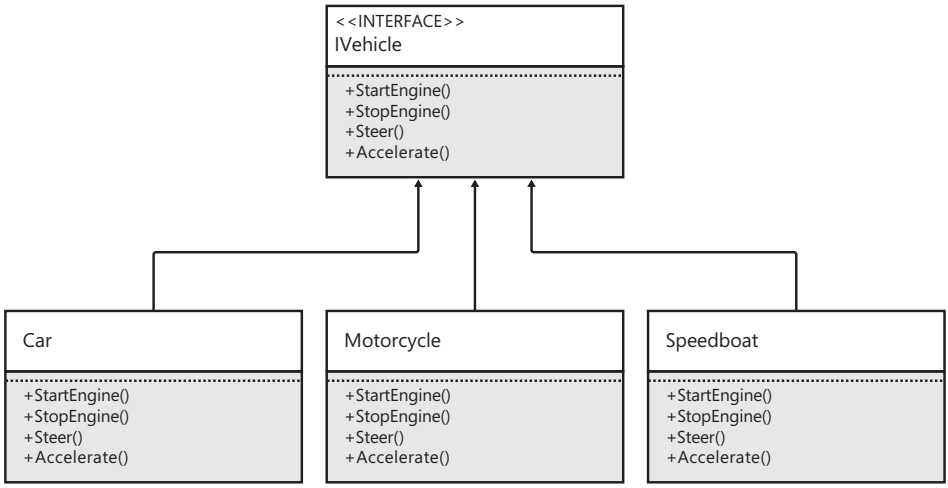

Abb. 3–3 Schnittstellen geben ihr Verhalten an die implementierenden Klassen weiter, was Polymorphie ermöglicht

In diesem Beispiel wird angenommen, dass Fahrzeuge einen Motor haben, der gestartet (`StartEngine`) und abgestellt (`StopEngine`) werden kann, dass es irgendeine Form der Lenkung gibt (`Steer`) und dass das Fahrzeug beschleunigen kann (`Accelerate`). Dank Polymorphie kann sich Clientcode einen Verweis auf die Schnittstelle `IVehicle` besorgen und alle spezialisierten Typen gleich behandeln. Dass ein Auto auf andere Weise gelenkt wird als ein Motorrad oder dass der Motor eines Boots anders gestartet oder abgestellt wird als der einer Lokomotive, ist für den Client unwichtig. Und das ist gut so. In der Praxis sind wir alle Clients dieser Schnittstelle, wenn wir ein Fahrzeug benutzen. Natürlich ist eine echte Schnittstelle für ein Fahrzeug komplexer als in diesem Beispiel, aber das Prinzip ist dasselbe. Müssen Sie wissen, wie der Motor in Ihrem Auto funktioniert, um ihn anzulassen oder abzustellen? Eindeutig Nein. Dies sind Implementierungsdetails, die Sie meist nicht benötigen, um das Fahrzeug zu benutzen. So sieht guter Schnittstellenentwurf aus.

Alle Entwurfs-Patterns und Schnittstellenfeatures, die in den weiteren Abschnitten dieses Kapitels vorgestellt werden, helfen beim Erstellen von adaptivem Code. Dank Polymorphie sind sie für jede Klasse nützlich, die eine benötigte Schnittstelle anbietet, unabhängig davon, ob sie bereits geschrieben ist oder erst noch entwickelt werden muss.

Adaptive Entwurfs-Patterns

Entwurfs-Patterns (engl. design patterns, auch als Entwurfsmuster bezeichnet) wurden durch das Gang-of-Four-Buch[1] *Software Patterns* (Addison-Wesley Professional, 1994; erste dt. Ausgabe *Entwurfsmuster*, Addison-Wesley, 1996) bekannt gemacht. Obwohl dieses Buch 20 Jahre alt ist (mindestens vier Eiszeiten in Softwareentwicklungsjahren), ist es auch heute noch relevant. Einige der Patterns werden zwar heute als Anti-Patterns eingestuft, aber andere werden ständig eingesetzt und verbessern die Anpassungsfähigkeit von Code.

Gute Entwurfs-Patterns sind wieder verwendbare Kombinationen aus Schnittstellen und Klassen, die in vielen unterschiedlichen Szenarien über Projekte, Plattformen, Sprachen und Frameworks hinweg angewendet werden können. Wie die meisten sinnvollen Empfehlungen sind auch Entwurfs-Patterns ein theoretisches Werkzeug, das Sie kennen sollten. Sie werden manchmal übertrieben eingesetzt und lassen sich nicht immer anwenden. Manchmal verkomplizieren sie eine andernfalls simple Lösung durch eine riesige Zahl von Klassen, Schnittstellen, Umleitung und Abstraktion.

Ich habe die Erfahrung gemacht, dass Entwurfs-Patterns entweder zu wenig oder zu viel genutzt werden. In manchen Projekten gibt es nicht genug Entwurfs-Patterns und der Code leidet unter einem Mangel an erkennbarer Struktur. Andere Projekte wenden Entwurfs-Patterns zu großzügig an und fügen Umleitungen und Abstraktion ein, wo das kaum einen Vorteil bringt. Die Kunst besteht darin, die richtigen Stellen zu finden und dort die richtigen Patterns anzuwenden.

1. Der Name leitet sich davon ab, dass vier Autoren beteiligt sind: Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides.

Das Null-Object-Pattern

Das Null-Object-Pattern ist eines der bekanntesten Entwurfs-Patterns. Außerdem ist es leicht zu verstehen und einfach zu implementieren. Es verhindert, dass Sie versehentlich eine Ausnahme des Typs `NullReferenceException` auslösen, und macht eine Menge Code überflüssig, der auf `null`-Objekte prüft. Das UML-Klassendiagramm in Abbildung 3–4 zeigt, wie das Null-Object-Pattern angewendet wird.

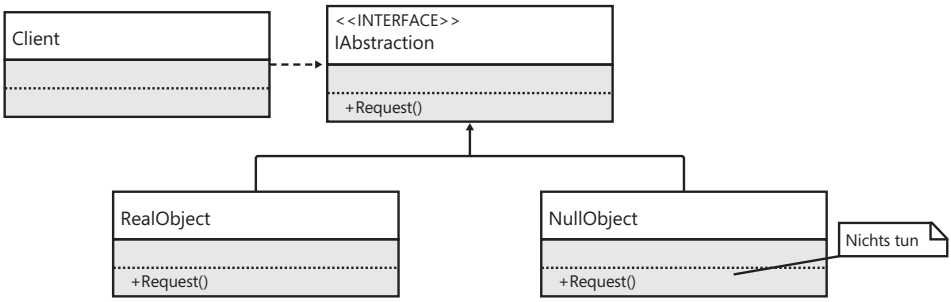

Abb. 3–4 Das Null-Object-Pattern als UML-Klassendiagramm

Listing 3–11 zeigt typischen Code, der eine `NullReferenceException` auslösen kann.

```
class Program
{
    static IUserRepository userRepository = new UserRepository();

    static void Main(string[] args)
    {
        var user = userRepository.GetByID(Guid.NewGuid());
        // Ohne Null-Object-Pattern könnte diese Zeile eine Ausnahme auslösen.
        user.IncrementSessionTicket();
    }
}
```

List. 3–11 Wenn Sie den Rückgabewert nicht auf `null` prüfen, besteht die Möglichkeit, eine `NullReferenceException` auszulösen

Jeder Client, der `IUserRepository.GetByID(Guid userID)` aufruft, läuft Gefahr, einen `null`-Verweis zu erhalten. In der Praxis bedeutet das, dass jeder Client prüfen muss, ob der Rückgabewert ungleich `null` ist. Ansonsten könnte beim Versuch, den `null`-Verweis zu benutzen, eine `NullReferenceException` ausgelöst werden. Der Clientcode sähe dann aus wie in Listing 3–12.

```
class Program
{
    static IUserRepository userRepository = new UserRepository();

    static void Main(string[] args)
    {
        var user = userRepository.GetByID(Guid.NewGuid());
        if(user != null)
        {
            user.IncrementSessionTicket();
        }
    }
}
```

List. 3–12 Soll wirklich *jeder* Client prüfen, ob er `null` zurückbekommt?

Das Prinzip des Null-Object-Patterns besagt in diesem Fall, dass Sie die Clients von `IUserRepository` zu stark belasten. Je mehr Clients die Methode verwenden, desto höher die Wahrscheinlichkeit, dass jemand vergisst, auf `null` zu prüfen. Stattdessen sollten Sie die Quelle des Problems so verändern, dass sie diese Überprüfung selbst erledigt. Das Ergebnis sehen Sie in Listing 3–13.

```
public class UserRepository : IUserRepository
{
    public UserRepository()
    {
        users = new List<User>
        {
            new User(Guid.NewGuid()),
            new User(Guid.NewGuid()),
            new User(Guid.NewGuid()),
            new User(Guid.NewGuid())
        };
    }

    public IUser GetByID(Guid userID)
    {
        IUser userFound = users.SingleOrDefault(user => user.ID == userID);
        if(userFound == null)
        {
            userFound = new NullUser();
        }
        return userFound;
    }

    private ICollection<User> users;
}
```

List. 3–13 Der Dienstcode sollte das Null-Object-Pattern implementieren

Dieser Code versucht zuerst, das `User`-Objekt in der Collection der vorhandenen Benutzer zu finden; hier hat sich noch nichts gegenüber der bisherigen Implementierung geändert. Anschließend wird aber geprüft, ob die zurückgegebene `User`-Instanz in Wirklichkeit ein `null`-Verweis ist. Ist das der Fall, wird ein Objekt einer speziellen Klasse zurückgegeben, die von `IUser` abgeleitet ist: `NullUser`. Diese abgeleitete Klasse überschreibt die Methode `IncrementSessionTicket` so, dass sie überhaupt nichts tut (Listing 3–14). Eine richtige `NullUser`-Implementierung überschreibt *alle* Methoden so, dass sie so wenig wie möglich tun.

```
public class NullUser : IUser
{
    public void IncrementSessionTicket()
    {
        // Nichts tun.
    }
}
```

List. 3–14 Die Methodenimplementierungen von `NullUser` tun überhaupt nichts

Außerdem sollte eine Methode oder Eigenschaft von `NullUser` immer dann, wenn von ihr erwartet wird, einen Verweis auf ein anderes Objekt zurückzugeben, eine spezielle Null-Object-Implementierung *dieser* Typen liefern. Anders ausgedrückt: Alle Null-Object-Implementierungen sollten rekursiv Null-Object-Implementierungen zurückgeben. Auf diese Weise brauchen die Clients überhaupt nicht mehr auf `null`-Verweise zu prüfen.

Das hat den zusätzlichen Vorteil, dass Sie weniger Unit-Tests schreiben müssen. Als vorher jeder Client die `null`-Prüfung implementieren musste, wurden entsprechende Unit-Tests gebraucht, um sicherzustellen, dass die Prüfung tatsächlich durchlaufen wurde. Stattdessen wird nun die Repository-Implementierung mit Unit-Tests darauf geprüft, ob sie eine `NullUser`-Implementierung zurückgibt.

Das IsNull-Property-Anti-Pattern

Manchmal erfordert es das Null-Object-Pattern, eine boolesche Eigenschaft namens `IsNull` zur Schnittstelle hinzuzufügen. Alle echten Implementierungen dieser Schnittstelle geben für diese Eigenschaft den Wert `false` zurück. Nur die Null-Object-Implementierung der Schnittstelle liefert den Wert `true`. Listing 3–15 zeigt auf Basis des vorherigen Beispiels, wie das funktioniert.

```
public interface IUser
{
    void IncrementSessionTicket();

    bool IsNull
    {
        get;
    }
}
```

```
// . . .
public class User : IUser
{
    // . . .
    public bool IsNull
    {
        get
        {
            return false;
        }
    }

    private DateTime sessionExpiry;
}
// . . .
public class NullUser : IUser
{
    public void IncrementSessionTicket()
    {
        // Nichts tun.
    }

    public bool IsNull
    {
        get
        {
            return true;
        }
    }
}
```

List. 3–15 Die Eigenschaft `IsNull` liefert nur für Null-Object-Implementierungen den Wert `true`

Das Problem bei dieser Eigenschaft besteht darin, dass Logik aus den Objekten entweicht, deren Zweck es eigentlich ist, Verhalten zu kapseln. Zum Beispiel werden `if`-Anweisungen im Clientcode auftauchen, um zwischen echten Implementierungen und der Null-Object-Implementierung zu unterscheiden. Das führt den ganzen Zweck des Patterns ad absurdum, das ja gerade verhindern sollte, die Prüflogik in die verschiedenen Clients zu tragen. Listing 3–16 zeigt ein typisches Beispiel für dieses Problem.

```
static void Main(string[] args)
{
    var user = userRepository.GetByID(Guid.NewGuid());
    // Ohne Null-Object-Pattern könnte diese Zeile eine Ausnahme auslösen.
    user.IncrementSessionTicket();

    string userName;
```

```
    if(!user.IsNull)
    {
        userName = user.Name;
    }
    else
    {
        userName = "unbekannt";
    }

    Console.WriteLine("Der Name des Benutzers ist {0}", userName);

    Console.ReadKey();
}
```

List. 3–16 Logik, mit der die Eigenschaft `IsNull` ausgewertet wird, macht dies zu einem Anti-Pattern

Das lässt sich leicht korrigieren, indem Sie den Namen eines `null`-Benutzers innerhalb der Klasse `NullUser` kapseln (Listing 3–17).

```
public class NullUser : IUser
{
    public void IncrementSessionTicket()
    {
        // Nichts tun.
    }

    public string Name
    {
        get
        {
            return "unbekannt";
        }
    }
}
// . . .
static void Main(string[] args)
{
    var user = userRepository.GetByID(Guid.NewGuid());
    // Ohne Null-Object-Pattern könnte diese Zeile eine Ausnahme auslösen.
    user.IncrementSessionTicket();

    Console.WriteLine("Der Name des Benutzers ist {0}", user.Name);

    Console.ReadKey();
}
```

List. 3–17 Bei durchdachter Kapselung wird die Eigenschaft `IsNull` obsolet

Cascading-Nulls-Operator

Als Erweiterung der Sprache C# wird gelegentlich gewünscht, ähnlich wie in Groovy[2] einen »Cascading Nulls«-Operator einzuführen. Sehen Sie sich dazu den folgenden Codeausschnitt an:

```
if(person != null && person.Address != null && person.Address.Country == "England")
{
    // . . .
}
```

Bei Einführung des neuen Sprachfeatures könnte dieser Code durch den folgenden ersetzt werden:

```
if(person?.Address?.Country == "England")
{
    // . . .
}
```

Der Operator `?.` wäre dann also eine Möglichkeit, jeden Objektverweis auf sichere Weise abzuarbeiten. Im schlimmsten Fall erhalten Sie den `default(T)` des Eigenschaftstyps zurück. Ich bin nicht grundsätzlich gegen Erweiterungen an der Syntax der Sprache, wenn andere sie nützlich finden, aber ich kann drei Argumente dagegen anführen, dies als Ersatz zu einer sorgfältigen Null-Object-Implementierung einzuführen.

Erstens gibt es zu viele Fälle, in denen der Standardwert eines Typs einfach nicht genügt. Das obige Beispiel, einen vernünftigen Benutzernamen zu liefern, damit keine `NullReferenceException` ausgelöst wird, zeigt, dass nicht ein Standardwert zurückgegeben werden sollte, sondern etwas, das für die Anwendung sinnvoll ist.

Zweitens müssten Sie dann beim Programmieren aller Clients dieses Codes die Möglichkeit eines `null`-Verweises im Hinterkopf behalten. Ein Anlass für die Verwendung des Null-Object-Patterns ist aber, die `null`-Prüfungen überflüssig zu machen und Ihnen zu ermöglichen, die Verweise unbesehen zu benutzen. Wenn Sie das Null-Object-Pattern durch die Cascading-Nulls-Syntax ersetzen, lauert wieder die Gefahr, dass jemand bei der Verweisauflösung schlampt.

Ein dritter, eher subjektiver Grund ist, dass eine solche Syntax wahrscheinlich überall Einzug halten würde. Wenn gelegentlich `int` mit Nullable-Option durch `int?` dargestellt wird, mag das ja noch akzeptabel sein, aber den Code bei allen Verweisauflösungen mit `?.` zu durchsetzen? Nein danke.

→

2. Groovy ist eine Java-Variante (*http://groovy-lang.org*) mit dynamischer Typisierung

Mit einer sinnvollen Null-Object-Implementierung können Sie dieses Beispiel folgendermaßen schreiben:

```
if(person.Address.Country == "England")
{
    // . . .
}
```

Das ist zweifellos die beste Lösung: Clientcode ohne Ballast, der dennoch vor den Gefahren einer `NullReferenceException` geschützt ist.

Das Adapter-Pattern

Mithilfe des Adapter-Patterns können Sie einem Client, der von einer Schnittstelle abhängig ist, eine passende Objektinstanz zur Verfügung stellen, obwohl Ihre Instanz diese Schnittstelle nicht implementiert. Dazu wird eine `Adapter`-Klasse erstellt, die die vom Client benötigte Schnittstelle bereitstellt, aber die Methoden dieser Schnittstelle dadurch implementiert, dass sie sie an unterschiedliche Methoden eines anderen Objekts delegiert. Dieses Pattern wird meist eingesetzt, wenn die Zielklasse nicht so geändert werden kann, dass sie die gewünschte Schnittstelle anbietet. Das kann den Grund haben, dass sie mit `sealed` markiert ist oder dass sie in einer Assembly liegt, deren Quellcode Sie nicht besitzen. Es stehen zwei Wege zur Wahl, das Adapter-Pattern zu implementieren: mit dem Class-Adapter-Pattern oder dem Object-Adapter-Pattern.

Das Class-Adapter-Pattern

Abbildung 3–5 zeigt, welche Klassen und Schnittstellen im Class-Adapter-Pattern zusammenarbeiten.

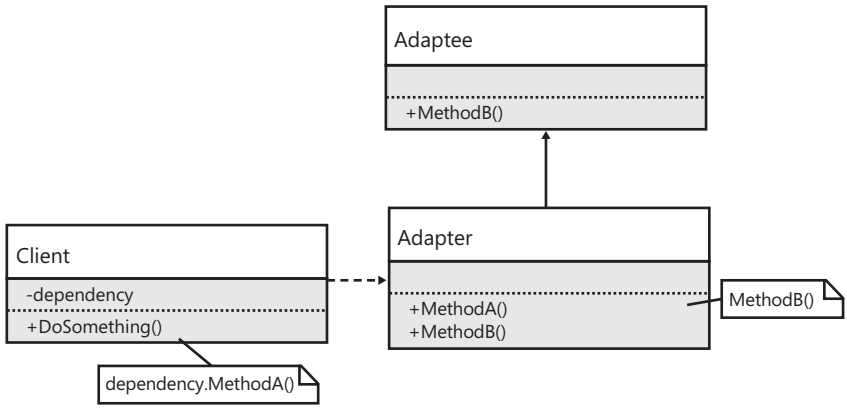

Abb. 3–5 Das UML-Klassendiagramm für das Class-Adapter-Pattern

Das Class-Adapter-Pattern nutzt Vererbung, um den Adapter zu erstellen. Es muss eine Klasse von der Zielklasse abgeleitet werden, um dem Client die benötigte Schnittstelle zur Verfügung stellen zu können. Listing 3–18 zeigt, wie das in der Praxis aussieht.

```
public class Adaptee
{
    public void MethodB()
    {

    }
}
// . . .
public class Adapter : Adaptee
{
    public void MethodA()
    {
        MethodB();
    }
}
// . . .
class Program
{
    static Adapter dependency = new Adapter();
    static void Main(string[] args)
    {
        dependency.MethodA();
    }
}
```

List. 3–18 Das Class-Adapter-Pattern arbeitet mit Implementierungsvererbung

Von den beiden Adapter-Patterns wird dieses seltener benutzt, vor allem weil Entwickler gepredigt bekommen, Komposition den Vorzug gegenüber der Vererbung zu geben. Das liegt daran, dass die abgeleitete Klasse bei der Vererbung von der Implementierung einer Klasse abhängig wird, nicht nur von ihrer Schnittstelle. Dagegen beschränkt sich die Abhängigkeit bei der Komposition auf die Schnittstelle, daher kann die Implementierung geändert werden, ohne dass dies Auswirkungen auf die Clients hätte.

Das Object-Adapter-Pattern

Das Object-Adapter-Pattern nutzt Komposition, um die Methoden der Schnittstelle an die eines intern gekapselten Objekts zu delegieren. Abbildung 3–6 zeigt, welche Klassen und Schnittstellen beim Object-Adapter-Pattern zusammenarbeiten.

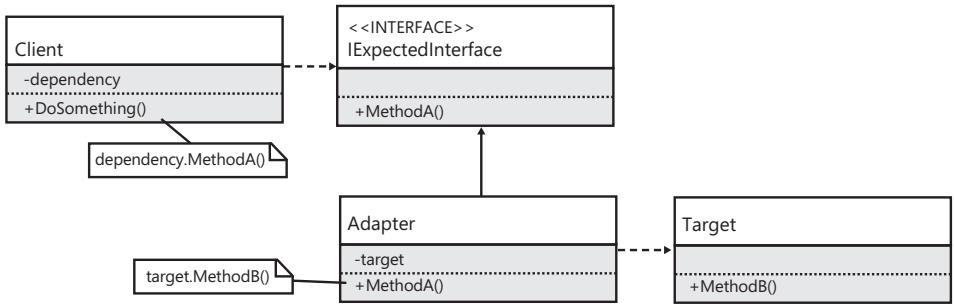

Abb. 3–6 Das UML-Klassendiagramm für das Object-Adapter-Pattern

Listing 3–19 zeigt, wie das Pattern in der Praxis aussieht.

```
public interface IExpectedInterface
{
    void MethodA();
}
// . . .
public class Adapter : IExpectedInterface
{
    public Adapter(TargetClass target)
    {
        this.target = target;
    }

    public void MethodA()
    {
        target.MethodB();
    }

    private TargetClass target;
}
//
public class TargetClass
{
    public void MethodB()
    {

    }
}
// . . .
class Program
{
    static IExpectedInterface dependency = new Adapter(new TargetClass());
    static void Main(string[] args)
    {
```

```
        dependency.MethodA();
    }
}
```

List. 3–19 Der Adapter bekommt die Zielklasse als Konstruktorargument übergeben und delegiert die Methoden dorthin

Das Strategy-Pattern

Mit dem Strategy-Pattern (»Strategie-Muster«) können Sie das Verhalten einer Klasse ändern, ohne sie neu zu kompilieren, unter Umständen sogar während der Laufzeit. Abbildung 3–7 zeigt das UML-Klassendiagramm für das Strategy-Pattern.

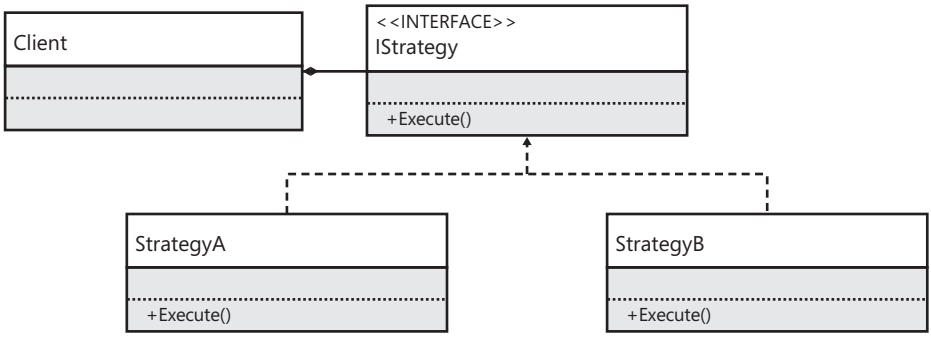

Abb. 3–7 Das UML-Klassendiagramm des Strategy-Patterns

Das Strategy-Pattern wird immer dann benutzt, wenn eine Klasse abhängig vom Zustand eines Objekts unterschiedliche Verhaltensweisen zeigen muss. Falls sich dieses Verhalten während der Laufzeit abhängig vom aktuellen Zustand der Klasse verändert, eignet sich das Strategy-Pattern perfekt, um dieses abweichende Verhalten zu kapseln. Listing 3–20 zeigt, wie Sie das Strategy-Pattern erstellen und in einer Klasse benutzen.

```
public interface IStrategy
{
    void Execute();
}
// . . .
public class ConcreteStrategyA : IStrategy
{
    public void Execute()
    {
        Console.WriteLine("ConcreteStrategyA.Execute()");
    }
}
// . . .
public class ConcreteStrategyB : IStrategy
{
```

```
    public void Execute()
    {
        Console.WriteLine("ConcreteStrategyB.Execute()");
    }
}
// . . .
public class Context
{
    public Context()
    {
        currentStrategy = strategyA;
    }

    public void DoSomething()
    {
        currentStrategy.Execute();

        // Strategie bei jedem Aufruf wechseln.
        currentStrategy = (currentStrategy == strategyA) ? strategyB : strategyA;
    }

    private readonly IStrategy strategyA = new ConcreteStrategyA();
    private readonly IStrategy strategyB = new ConcreteStrategyB();

    private IStrategy currentStrategy;
}
```

List. 3–20 Das Strategy-Pattern

Bei jedem Aufruf von `Context.DoSomething` delegiert die Methode die Aktion an die aktuelle Strategie weiter und wechselt dann zwischen Strategie A und Strategie B. Beim nächsten Aufruf delegiert sie die Aktion an die neu ausgewählte Strategie, bevor sie wieder zur ursprünglichen Strategie zurückwechselt.

Wie die Strategien ausgewählt werden, ist ein Implementierungsdetail, auf den eigentlichen Zweck des Patterns hat das keinen Einfluss: Das Verhalten der Klasse wird hinter einer Schnittstelle versteckt, deren Implementierungen benutzt werden, um die tatsächliche Arbeit zu erledigen.

Mehr Vielseitigkeit

Der Nutzen von Schnittstellen beschränkt sich nicht auf Entwurfs-Patterns. Es gibt andere, noch weiter spezialisierte Nutzungen von Schnittstellen, die eine genauere Betrachtung verdienen. Diese Features sind zwar nicht überall einsetzbar, aber es gibt Situationen, in denen sie das passende Werkzeug für die Aufgabe sind.