

Die Kraft von C# voll ausschöpfen!

**Deutsche
Ausgabe**

C# 3.0

Entwurfsmuster



O'REILLY®

Judith Bishop
Deutsche Übersetzung von Thomas Demmig

C# 3.0 Entwurfsmuster

C# 3.0 Entwurfsmuster

Judith Bishop

*Deutsche Übersetzung von
Thomas Demmig*

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.

Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller. Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Alle Rechte vorbehalten einschließlich der Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Kommentare und Fragen können Sie gerne an uns richten:

O'Reilly Verlag
Balthasarstr. 81
50670 Köln
Tel.: 0221/9731600
Fax: 0221/9731608
E-Mail: kommentar@oreilly.de

Copyright der deutschen Ausgabe:
© 2008 by O'Reilly Verlag GmbH & Co. KG
1. Auflage 2008

Die Originalausgabe erschien 2008 unter dem Titel
C# 3.0 Design Patterns bei O'Reilly Media, Inc.

Die Darstellung von Gänsen im Zusammenhang mit dem
Thema C# und Entwurfsmustern ist ein Warenzeichen von O'Reilly Media, Inc.

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte bibliografische Daten
sind im Internet über <http://dnb.ddb.de> abrufbar.

Übersetzung und deutsche Bearbeitung: Thomas Demmig
Lektorat: Volker Bombien
Korrektur: Eike Nitz
Satz: III-satz, Husby
Umschlaggestaltung: Karen Montgomery, Sebastopol & Michael Oreal, Köln
Produktion: Andrea Miß, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-89721-867-3

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Vorwort	VII
Einleitung	XI
1 C# trifft auf Entwurfsmuster	1
Über Muster	3
Über UML	4
Über C# 3.0	6
Zu den Beispielen	7
2 Strukturelle Muster: Dekorierer, Proxy und Brücke	9
Dekorierer-Muster	11
Proxy-Muster	24
Brücken-Muster	39
Beispiel: OpenBook	43
Vergleich der Muster	50
3 Strukturmuster: Kompositum und Fliegengewicht	53
Kompositum-Muster	53
Fliegengewicht-Muster	65
Übungen	77
Vergleich der Muster	78
4 Strukturmuster: Adapter und Fassade	81
Adapter-Muster	81
Fassade-Muster	101
Vergleich der Muster	108
5 Erzeugungsmuster: Prototyp, Fabrikmethode und Singleton	111
Prototyp-Muster	111
Fabrikmethoden-Muster	121

Singleton-Muster	126
Vergleich der Muster	131
6 Erzeugungsmuster: Abstrakte Fabrik und Erbauer	135
Abstrakte-Fabrik-Muster	135
Erbauer-Muster	143
Vergleich der Muster	151
7 Verhaltensmuster: Strategie, Zustand und Schablonenmethode	153
Strategie-Muster	153
Zustand-Muster	163
Schablonenmethode-Muster	173
Vergleich der Muster	178
8 Verhaltensmuster: Zuständigkeitskette und Befehl	179
Zuständigkeitskette-Muster	179
Befehl-Muster	191
Vergleich der Muster	203
9 Verhaltensmuster: Iterator, Vermittler und Beobachter	205
Iterator-Muster	205
Vermittler-Muster	218
Beobachter-Muster	229
Besprechung und Vergleich der Muster	237
10 Verhaltensmuster: Besucher, Interpreter und Memento	241
Besucher-Muster	241
Interpreter-Muster	255
Memento-Muster	264
Vergleich der Muster	274
11 Die Zukunft der Entwurfsmuster	277
Zusammenfassung der Muster	277
Die Zukunft der Entwurfsmuster	280
Abschließende Bemerkungen	282
Anhang	283
Literatur	307
Index	309

Strukturmuster: Kompositum und Fliegengewicht

Die Strukturmuster Kompositum und Fliegengewicht sind für Systeme nützlich, die viele Datenobjekte enthalten. Das Kompositum-Muster lässt sich sehr häufig anwenden und seine Kompositum-Listen können zudem das Fliegengewicht-Muster nutzen. Das Fliegengewicht-Muster behandelt identische Objekte im Hintergrund gemeinsam, um Platz zu sparen. Bei der Implementierung nutzen diese Muster die folgenden neuen Features von C#:

- Generics
- Eigenschaften
- Structs
- Indexer
- Implizite Typisierung
- Initialisierer
- Anonyme Typen

Kompositum-Muster

Rolle

Das Kompositum-Muster (Composite) ordnet strukturierte Hierarchien so, dass einzelne Komponenten und Gruppen von Komponenten auf die gleiche Art und Weise behandelt werden können. Typische Operationen auf diesen Komponenten sind Hinzufügen, Entfernen, Anzeigen, Finden und Gruppieren.

Illustration

Computer-Anwendungen, die sich auf das Gruppieren von Daten spezialisiert haben, sind heutzutage überall zu finden. Schauen Sie sich nur eine Musik-Verwal-

tung wie in iTunes oder ein digitales Fotoalbum wie Flickr oder iPhoto an (Abbildung 3-1). Elemente werden in eine große Liste gesteckt, die danach strukturiert wird.



Abbildung 3-1: Illustration des Kompositum-Musters – iPhoto

Schauen wir uns den Screenshot von iPhoto an, können wir erkennen, dass es verschiedene Möglichkeiten gibt, die importierten Fotos zu betrachten: chronologisch oder anhand von Ereignissen. Ein einzelnes Foto kann in vielen Alben auftauchen (zum Beispiel »Last Roll«, »2007« und »Switzerland«). Erstellt man ein Album, erzeugt man ein Kompositum-Objekt, kopiert damit aber nicht die Fotos. In diesem Kontext ist das entscheidende am Kompositum-Muster, dass die auf Fotos und Fotoalben ausgeführten Operationen die gleichen Namen und Effekte haben sollten, auch wenn sich die Implementierungen unterscheiden. So sollte der Benutzer zum Beispiel in der Lage sein, ein Foto oder ein Album (mit Fotos) anzeigen zu können. Genauso sollte das Löschen eines einzelnen Fotos oder eines Albums auf die gleiche Art und Weise geschehen.

Design

Das Kompositum-Muster ist oberflächlich betrachtet eines der einfachsten unter den Mustern. Es muss mit zwei Typen umgehen: *Components* und *Composites* dieser Komponenten. Beide Typen besitzen eine Schnittstelle mit gemeinsamen Operationen. *Composite*-Objekte bestehen aus *Components* und in den meisten Fällen werden Operationen auf einem *Composite* implementiert, indem die entsprechenden Opera-

tionen für die enthaltenen Component-Objekte aufgerufen werden. Siehe dazu auch Abbildung 3-2 mit dem UML-Diagramm für dieses Muster.

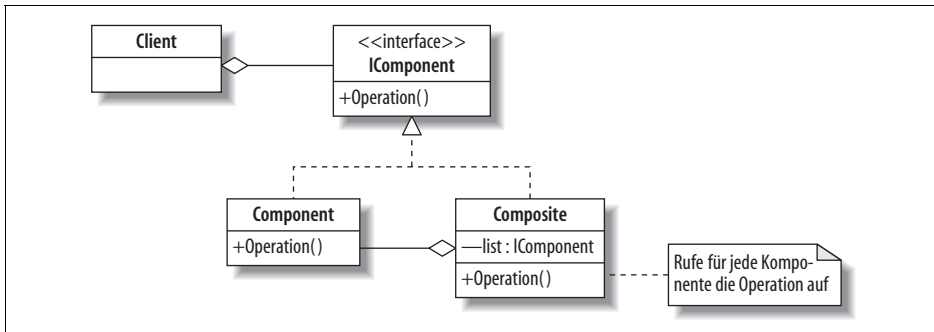


Abbildung 3-2: UML-Diagramm für das Kompositum-Muster

Die wichtigsten Teilnehmer im UML-Diagramm des Kompositum-Musters sind:

IComponent

Definiert die Operationen für Objekte im Kompositum und jedes Standard-Verhalten, dass für Objekte beider Typen passt.

Operation

Führt eine Operation auf Objekten durch, die zu IComponent konform sind.

Component

Implementiert die Operationen auf einzelnen Objekten, die nicht weiter auseinandergenommen werden können.

Composite

Implementiert die Operationen auf kombinierten Objekten, wobei eine lokal vorgehaltene Liste von Komponenten genutzt wird, auf die die entsprechenden Operationen von Component angewandt werden.

Der Client arbeitet nur mit dem Interface IComponent, womit es für ihn einfacher wird.

Implementierung

Auch wenn sich unsere Illustration auf Fotos und Alben bezieht, ist das Design und die Implementierung des Kompositum-Musters vollständig unabhängig von der Art der Basiskomponente, die organisiert werden soll. Wir können genauso mit Gruppen von Personen oder Bankkonten-Portfolios arbeiten. Die Kompositum-Operationen müssen allerdings über eine Struktur dieser Komponenten iterieren. Um sowohl die Flexibilität durch Komponenten beliebigen Typs, als auch eine Verbindung zwischen Blatt und Kompositum zu ermöglichen, können wir in C# auf *Generics* zurückgreifen.

QUIZ

Verbinde die Teilnehmer des Kompositum-Musters mit der iPhoto-Illustration

Um zu testen, ob Sie das Kompositum-Muster verstanden haben, decken Sie die linke Spalte der unteren Tabelle mit der Hand ab und versuchen Sie, die Teilnehmer anhand des Beispiels (Abbildung 3-1) in der rechten Spalte zuzuordnen. Dann vergleichen Sie Ihre Antworten mit der linken Spalte.

IComponent	Eine sichtbare Entität in iPhoto
Component	Ein einzelnes Foto
Composite	Ein Fotoalbum
Operation	Öffnen und Anzeigen

C#-Feature – Generics

Generics sind eine Erweiterung des Typ-Systems, mit der Structs, Klassen, Interfaces, Delegates und Methoden durch Typen parametrisiert werden können. So ist zum Beispiel ein generischer Typ wie `List<T>` ein Generator für »richtige« Typen wie `List<string>` und `List<Person>`. Alle Collections in der .NET-Framework-Bibliothek, die von C# genutzt werden, stehen in generischer Form zur Verfügung. Dazu gehören unter anderem `Dictionary`, `Stack`, `Queue` und `List`, sowie Variationen davon. Es gibt auch generische Methoden, wie `Sort` und `BinarySearch`. Dabei ist es erforderlich, dass die Typen, mit denen sie arbeiten sollen, das Interface `IComparer` implementieren, so dass Vergleiche zwischen den Elementen korrekt vorgenommen werden können.

Um einen generischen Typ oder eine Methode zu spezifizieren, geben Sie den/die generischen Parameter in spitzen Klammern an, wie zum Beispiel `<T>` oder `<T, P>`. Der generische Typ kann genutzt werden, um weitere generische Elemente zu spezifizieren, wobei wieder das Format `<T>` zu verwenden ist, oder er kann ganz normal als Typ `T` verwendet werden.

Um aus einer generischen Methode oder einem generischen Typ einen »richtigen« zu machen, stellen Sie echte Typen für jeden der generischen Parameter bereit, zum Beispiel `<string>`.

Siehe auch C# Language Specification Version 3.0, September 2007, Section 10.1.3

Indem wir `IComponent`, `Component` und `Composite` als generische Typen deklarieren, erzeugen wir eine Implementierung des Kompositum-Musters, das in einem Praxis-Beispiel genutzt werden kann.



Für das Kompositum-Muster erzeugen wir kein *Programm*, sondern einen *Namensraum* mit den zwei Klassen *Composite* und *Component* sowie einem Interface *IComponent*.

Beginnen wir mit dem Interface *IComponent*. Wir deklarieren es als generisch vom Typ *T*. Dann folgt jede der Operationen:

```
public interface IComponent <T> {  
    void        Add(IComponent <T> c);  
    IComponent <T> Remove(T s);  
    IComponent <T> Find(T s);  
    string      Display(int depth);  
    T           Item {get; set;}  
}
```



Da sich die Typen in einem eigenen Namensraum befinden, müssen sie alle als *public* deklariert werden.

C# 3.0-Feature – Eigenschaften und Accessors

Eine *Eigenschaft* ermöglicht einen kontrollierten Zugriff auf den privaten, lokalen Status eines Objekts – entweder direkt auf seine Felder oder über irgendwelche Berechnungen. Eigenschaften definieren einen oder zwei Accessors: *get* und/oder *set*. Die häufigste Form einer Eigenschaft ist:

```
public type Field {get; set;}
```

Dabei ist *Field* ein Bezeichner. Diese Deklaration erzeugt einen privaten Member und macht ihn für die Öffentlichkeit zum Lesen und Schreiben über den Namen *Field* erreichbar.

Indem solche Eigenschaften in einem Interface untergebracht werden, nötigen wir die implementierenden Klassen im Endeffekt, die Eigenschafts-Accessors neben den normalen Methoden ebenfalls zu implementieren.

Eigenschaften können entweder den *get*- oder den *set*-Accessor weglassen – häufig lässt man *set* weg, womit die Eigenschaft (von außen) schreibgeschützt ist.

Eigenschaften können das Ergebnis eines *get* oder *set* auch berechnen, dann ist die Syntax etwas erweitert:

```
public type Field {  
    get { Anweisungen; return Ausdruck; }  
    set { Anweisungen, die eine Referenz auf value enthalten; }  
}
```

Eine Eigenschaft hat normalerweise (aber nicht zwingend) Zugriff auf ein *private type Field*. *value* ist der implizite Parameter beim Setzen einer Eigenschaft.

Siehe auch C# Language Specification Version 3.0, September 2007, Section 10.7

Warum beziehen sich Add, Remove und Find auf IComponent <T>, Remove und Find aber auch noch auf T? Was ist der Unterschied zwischen <T> und T? Denken Sie daran, dass ein Objekt vom Typ IComponent entweder ein Component oder ein Composite ist, zudem aber auch ein IComponent – neben dem eigentlichen Wert des Elements, dass das Objekt speichert. Das Element ist vom Typ T, daher übergeben wir bei Find und Remove als Parameter die eigentlichen Werte vom Typ T. Wenn das Interface zum Beispiel mit T als Image instanziiert worden wäre würden wir ein Image-Objekt an Find übergeben und im Gegenzug eine Referenz auf ein Objekt vom Typ IComponent erhalten. Das zurückgegebene Objekt würde das Image-Objekt enthalten, wenn unsere Suche erfolgreich gewesen war.

Die letzte Zeile des Interface ist interessant, da dort eine neue Syntax für Eigenschaften in Klassen genutzt wird. Vor C# 3.0 gab es diese Syntax nur für Interfaces. Sie wurde nun auch auf Klassen ausgeweitet und macht Programme damit kürzer und leichter lesbar.

Lassen Sie uns nun schauen, wie die Klasse Component unter Berücksichtigung des Interface IComponent implementiert werden kann. Das geschieht im Namensraum in Beispiel 3-1.

Beispiel 3-1: Namensraum-Code für das Kompositum-Muster

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text; // für StringBuilder
4
5 namespace CompositePattern {
6
7     // Das Interface
8     public interface IComponent <T> {
9         void Add(IComponent <T> c);
10        IComponent <T> Remove(T s);
11        string Display(int depth);
12        IComponent <T> Find(T s);
13        T Name {get; set;}
14    }
15
16    // Die Komponente
17    public class Component <T> : IComponent <T> {
18        public T Name {get; set;}
19
20        public Component (T name) {
21            Name = name;
22        }
23
24        public void Add(IComponent <T> c) {
25            Console.WriteLine("Kann einem Element nichts hinzufügen");
26        }
27
28        public IComponent <T> Remove(T s) {
```

Beispiel 3-1: Namensraum-Code für das Kompositum-Muster (Fortsetzung)

```
29     Console.WriteLine("Kann nicht direkt entfernen");
30     return this;
31 }
32
33 public string Display(int depth) {
34     return new String('-', depth) + Name+"\n";
35 }
36
37 public IComponent <T> Find (T s) {
38     if (s.Equals(Name))
39         return this;
40     else
41         return null;
42 }
43 }
44
45 // Das Kompositum
46 public class Composite <T> : IComponent <T> {
47     List <IComponent <T>> list;
48
49     public T Name {get; set;}
50
51     public Composite (T name) {
52         Name = name;
53         list = new List <IComponent <T>> ();
54     }
55
56     public void Add(IComponent <T> c) {
57         list.Add(c);
58     }
59
60     IComponent <T> holder=null;
61
62     // Finde das Element ausgehend von einem Punkt in der Struktur
63     // und liefere das Kompositum zurück, aus dem es entfernt wurde
64     // Wenn nicht gefunden, gib den Punkt zurück
65     public IComponent <T> Remove(T s) {
66         holder = this;
67         IComponent <T> p = holder.Find(s);
68         if (holder!=null) {
69             (holder as Composite<T>).list.Remove(p);
70             return holder;
71         }
72         else
73             return this;
74     }
75
76     // Suche rekursiv nach einem Element
77     // Liefere seine Referenz oder null zurück
78     public IComponent <T> Find (T s) {
79         holder = this;
```

Beispiel 3-1: Namensraum-Code für das Kompositum-Muster (Fortsetzung)

```
80     if (Name.Equals(s)) return this;
81     IComponent <T> found=null;
82     foreach (IComponent <T> c in list) {
83         found = c.Find(s);
84         if (found!=null)
85             break;
86     }
87     return found;
88 }
89
90 // Zeige Elemente in einem Format an, das ihre Position in der Struktur
91 //darstellt
92 public string Display(int depth) {
93     StringBuilder s = new StringBuilder(new String('-', depth));
94     s.Append("Set "+ Name + " Größe: " + list.Count + "\n");
95     foreach (IComponent <T> component in list) {
96         s.Append(component.Display(depth + 2));
97     }
98     return s.ToString();
99 }
100 }
```

Der Namensraum enthält zunächst die Definition des Interface mit seinen vier Methoden und einer Eigenschaft. Dann kommt die Klasse `Component`. Nicht alle Methoden von `IComponent` sind für `Component` wichtig. Das Hinzufügen und Entfernen geschieht nur in `Composites`, daher wird eine einfache Fehlermeldung ausgegeben (siehe den Abschnitt »Übungen« für ein Erweitern dieses Punktes). In der Methode `Find` von `Component` (Zeilen 37–42), verlässt sich die Klasse darauf, dass der tatsächlich genutzte Typparameter für `T` eine Methode `Equals` besitzt. Wenn das nicht der Fall ist, wird die generische Instanziierung beim Kompilieren einen Fehler melden.

Die Methode `Display` (Zeilen 33–35) geht ebenfalls davon aus, dass das Feld, das über die Eigenschaft `Name` angesprochen wird, eine Methode `ToString` besitzt. Tatsächlich werden hier nur String-ähnliche Typen ordentlich funktionieren, daher ist es wahrscheinlich, dass die Methode `Display` später definiert werden sollte (siehe den Abschnitt »Übungen«).

Die Klasse `Composite` implementiert ebenfalls das Interface `IComponent` (Zeile 46). Die Methoden `Find` und `Remove` von `Composite` sind umfassender als die in `Component`, damit sie beliebige Strukturen aus Komposita und Komponenten handhaben können. Lassen Sie uns ein paar der interessanteren Anweisungen genauer anschauen:

- `Composite` enthält als Liste eine lokale Struktur, die aus `Components` und `Composites` besteht (Zeile 47). Wenn es sich beim Inhalt um `Composites` han-

delt, wird ein neues Objekt und eine neue Liste erstellt. Die Liste ist wie folgt deklariert:

```
List <IComponent <T>> list;
```

Man sieht, dass ein offener, generischer Typ als Parameter für einen anderen generischen Typ verwendet werden kann.

- In `Remove` (Zeilen 65–74) suchen wir zunächst in der Struktur nach dem Element und entfernen es – falls gefunden – aus der Listenstruktur, die lokal in `Composite` gehalten wird (Zeile 69):

```
(holder as Composite<T>).list.Remove(p);
```

Die Variable `holder` ist vom Typ `IComponent` und muss in ein `Composite` gecastet werden, bevor man auf die Liste zugreifen kann.

- Ein offener, generischer Typ kann auch in einer `foreach`-Schleife genutzt werden, wie in den Methoden `Find` und `Display` zu sehen (Zeilen 82 und 94):

```
foreach (IComponent <T> c in list) {  
    found = c.Find(s);  
}
```

- Der Aufruf von `Find` geschieht für die passende Methode, die davon abhängt, was der tatsächliche Typ von `c` zur Laufzeit ist. Das ist ideal für das Kompositum-Muster, das `Component` und `Composite` gleich behandelt sehen möchte.

Damit ist die theoretische Implementierung des Kompositum-Musters abgeschlossen. Abgesehen von den Bedenken bezüglich der Methode `Display` können die drei definierten Typen zusammen für beliebige Elemente verwendet werden. Daher stecken sie im Namensraum `CompositePattern` und werden im nächsten Beispiel verwendet.

Beispiel: Foto-Bibliothek

In diesem Beispiel geht es uns darum, die Dateinamen digitaler Fotos in mit Namen versehenen Sets zu sammeln. Wir werden in diesem Beispiel nicht die eigentlichen Bilder nutzen, sondern nur die Dateinamen als Strings. Der Client erhält eine Domänen-spezifische Sammlung von Befehlen, mit denen er die Bibliothek erzeugen und verwalten kann. Aus Sicht des Benutzers ist ein zentraler Punkt beim Bearbeiten der Bibliothek die Frage: »Wo bin ich?« Wir beginnen mit einem leeren Set namens »Album«. Einige der Befehle belassen das System auf der Komponente, die ausgewählt war, während andere zurück zur ersten Komponente im Set springen. Die Befehle sind:

`AddSet`

Fügt ein neues, leeres Set mit einem Namen hinzu und verbleibt dort.

`AddPhoto`

Fügt ein neues, mit Namen versehenes Foto hinter dem Zeiger hinzu und verbleibt dort.

Find

Findet die per Namen definierte Komponente (Set oder Foto) oder liefert null zurück, wenn sie nicht gefunden wurde.

Remove

Entfernt die per Namen definierte Komponente (Set oder Foto) und verbleibt in dem Set, aus dem sie entfernt wurde.

Display

Zeigt die gesamte Struktur an.

Quit

Verlässt das Programm.

Damit sind die beiden Operationen, die sowohl auf Components als auch auf Composites arbeiten, Find und Remove. Schauen Sie sich ein paar Beispiele für die Arbeit mit diesem System in Beispiel 3-2 an. Die Eingabebefehle sind mittig ausgegeben, das Ergebnis von Display linksbündig und ein paar Anmerkungen finden sich rechts.



Die Eingabedatei enthält alle Befehle in der Mitte. Das Programm erwartet diese Befehle in einer Datei unter dem Namen *Composite.dat*.

Beispiel 3-2: Kompositum-Muster – Beispiellauf für die Foto-Bibliothek

```
AddSet Home
AddPhoto Dinner.jpg
AddSet Pets      Zu einer andere Ebene wechseln
AddPhoto Dog.jpg
AddPhoto Cat.jpg
Find Album      Dafür sorgen, dass Garden auf Home-Ebene ist
AddSet Garden
AddPhoto Spring.jpg
AddPhoto Summer.jpg
AddPhoto Flowers.jpg
AddPhoto Trees.jpg
Display          An den Anfang von Album springen

Set Album Größe: 2
--Set Home Größe: 2
----Dinner.jpg
----Set Pets Größe: 2
-----Dog.jpg
-----Cat.jpg
--Set Garden Größe: 4
----Spring.jpg
----Summer.jpg
----Flowers.jpg
----Trees.jpg
Remove Flowers.jpg      In Garden bleiben
AddPhoto BetterFlowers.jpg  Am Ende von Garden
```

Beispiel 3-2: Kompositum-Muster – Beispiellauf für die Foto-Bibliothek (Fortsetzung)

```
        Display
Set Album Größe: 2
--Set Home Größe: 2
----Dinner.jpg
----Set Pets Größe: 2
-----Dog.jpg
-----Cat.jpg
--Set Garden Größe: 4
----Spring.jpg
----Summer.jpg
----Trees.jpg
----BetterFlowers.jpg
        Find Home
        Remove Pets
        Display
Set Album Größe: 2
--Set Home Größe: 1
----Dinner.jpg
--Set Garden Größe: 4
----Spring.jpg
----Summer.jpg
----Trees.jpg
----BetterFlowers.jpg
        Quit
```

Die Klasse Client, die diese Befehle implementiert, ist in Beispiel 3-3 zu sehen. Es handelt sich um einen einfachen Befehls-Interpreter, der sich die Möglichkeit von C# zu Nutze macht, ein switch mit Strings vorzunehmen.

Beispiel 3-3: Beispiel-Code für das Kompositum-Muster – Foto-Bibliothek

```
using System;
using System.Collections.Generic;
using System.IO;
using CompositePattern;

// Der Client
class CompositePatternExample {
    static void Main () {
        IComponent <string> album = new Composite<string> ("Album");
        IComponent <string> point = album;
        string [] s;
        string command, parameter;
        // Erzeuge und Bearbeite eine Struktur
        StreamReader instream = new StreamReader("Composite.dat");
        do {
            string t = instream.ReadLine();
            Console.WriteLine("\t\t\t\t"+t);
            s = t.Split();
            command = s[0];
            if (s.Length>1) parameter = s[1]; else parameter = null;
            switch (command) {
```

```
        case "AddSet" :
            IComponent <string> c = new Composite <string> (parameter);
            point.Add(c);
            point = c;
            break;
        case "AddPhoto" :
            point.Add(new Component <string> (parameter));
            break;
        case "Remove" :
            point = point.Remove(parameter);
            break;
        case "Find" :
            point = album.Find(parameter);
            break;
        case "Display" :
            Console.WriteLine(album.Display(0));
            break;
        case "Quit" :
            break;
    }
} while (!command.Equals("Quit"));
}
```

Verwendung

Das Kompositum-Muster lässt sich an vielen Stellen anwenden und wird häufig zusammen mit dem Dekorierer-, Iterator- und Besucher-Muster genutzt. Seine Kompositum-Liste kann zudem das Fliegengewichts-Muster verwenden, das als nächstes behandelt wird. Das Kompositum-Muster sieht wie die Implementierung einer normalen Datenstruktur aus, ist aber mehr als das, da es die Möglichkeit bietet, die verschiedenen Arten von Elementen gleich zu behandeln.

Verwenden Sie das Kompositum-Muster, wenn ...

Sie haben:

- Eine unregelmäßige Struktur aus Objekte und Gruppen dieser Objekte.

Sie wollen:

- Clients sollen sich nur um die Unterschiede zwischen einzelnen Objekten und Kompositionen dieser Objekte kümmern müssen.
- Alle Objekte in einer Komposition sollen gleich behandelt werden.

Aber bedenken Sie dabei:

- Das Dekorierer-Muster, wenn Sie Operationen wie Add, Remove und Find bereitstellen.
- Das Fliegengewicht-Muster, um Komponenten gemeinsam zu verwalten, wenn das Wissen über die eigene Position unwichtig ist und alle Operationen am Wurzelknoten der Komposition ansetzen können.
- Das Besucher-Muster, um die Operationen zu sammeln, die momentan zwischen den Klassen Composite und Component aufgeteilt sind.

Übungen

1. Ein Manager ist ein Mitarbeiter, der Entwickler, Techniker und Support-Personal leitet, die alle ebenfalls Mitarbeiter sind. Modellieren Sie dieses Szenario mit dem Kompositum-Muster und konzentrieren Sie sich dabei auf die Operation »In Urlaub gehen«.
2. Im Beispiel für das Kompositum-Muster hat die Klasse `Component` zwei Fehlerbedingungen. Implementieren Sie sie mit Hilfe von Exceptions und entscheiden Sie, ob die Exceptions Teil des Interface `IComponent` sein sollen. Wie sollten Ihrer Meinung nach Muster im Allgemeinen mit Exceptions umgehen?
3. Die `Display`-Methoden von `Component` und `Composite` gehen davon aus, dass sich der Typ `T` wie ein String verhält. Erstellen Sie eine Version des Kompositum-Musters, in der `T` kein String-ähnlicher Typ ist (sondern zum Beispiel vom Typ `Image`) und untersuchen Sie, ob die `Display`-Methoden durch Extension-Methoden im Namensraum von `Client` überschrieben werden können.
4. Ändern Sie in der Klasse `Composite` die `List` in ein `Dictionary` um. Wie beeinflusst dies die Operation `Find`?
5. Auch wenn wir versuchen, ein Interface für Komponenten und Kompositionen zu haben, gibt es einen alternativen Ansatz: nur die Operationen aufnehmen, die wirklich gemeinsam nutzbar sind, und diejenigen separieren, die zum Beispiel nur für Komposita gültig sind. Wir haben dann zwei Interface-Ebenen, zum Beispiel:

```
interface IComponent {
    // Name, Display
}
interface IComposite<T> IComponent where T : IComponent {
    // Add, Remove, Find
}
```

Nun muss `IComponent` nicht mehr generisch sein – eine Komponente ist einfach etwas, das einen Namen hat und angezeigt werden kann. Zudem fallen die Probleme weg, wann `T` und wann `IComponent<T>` zu nutzen sind. Programmieren Sie das Beispiel auf diese Weise um. (Hinweis: das zweite oben angegebene Interface nutzt generische Constraints, die in Kapitel 6 behandelt werden.)

Fliegengewicht-Muster

Rolle

Das Fliegengewicht-Muster (Flyweight) bietet einen effizienten Weg, gemeinsame Informationen auch gemeinsam zu verwalten, wenn sich diese in kleinen, aber in einem System reichhaltig vorhandene Objekten befinden. Damit wird dabei geholfen, den Speicherbedarf zu reduzieren, der ansonsten bei vielen mehrfach vorhande-

nen Werten erforderlich wäre. Das Fliegengewicht-Muster unterscheidet zwischen dem *intrinsischen* und *extrinsischen* Zustand eines Objekts. Die größten Einsparungsmöglichkeiten ermöglicht das Fliegengewicht-Muster, wenn Objekte beide Zustandsarten nutzen:

- Der intrinsische Zustand kann in großem Umfang gemeinsam verwaltet werden, was Speicheranforderungen minimiert.
- Der extrinsische Zustand kann bei Bedarf berechnet werden, wobei man den Rechenaufwand gegen den Speicheraufwand abwägen muss.

Illustration

Wir werden uns nun der Frage der Bilddarstellung für die Foto-Bibliothek zuwenden, die im Kompositum-Muster behandelt wurde. Wir wollen jederzeit eine ganze Seite mit Bildern angezeigt haben und ohne große Zeitverzögerung durch die Bibliothek blättern können. Das bedeutet, dass so viele Bilder wie möglich im Voraus in den Speicher geladen und dort vorgehalten werden sollten, während die Foto-Anwendung läuft. Für eine Foto-Gruppierungs-Anwendung ist die Hauptfunktion, Fotos in Gruppen zu ordnen. Fotos können zu vielen verschiedenen Gruppen gehören, daher kann die Zahl der anzuzeigenden Bilder sehr groß werden. Wenn nicht alle Bilder in den Speicher passen, bedeutet die Tatsache, dass sie zu verschiedenen Gruppen gehören können, einen unregelmäßigen und nicht gut vorhersagbaren Bedarf bei der Anzeige, was zu vielen Festplattenzugriffen führen kann.

Schauen Sie sich die Anwendung in Abbildung 3-3 an. Bei einem kleineren Fenster könnte es sein, dass die ersten beiden Gruppen schon aus dem Fenster herausgeschoben wurden, wenn die Food-Gruppe erscheint, was bedeutet, dass mindestens drei der Bilder neu geladen und angezeigt werden müssten.

Der einem Objekt eigene, *ungeteilte* Zustand ist das Set von Gruppen, zu denen es gehört. Sein extrinsischer Zustand ist das eigentliche Bild, was groß sein kann – es beansprucht über 2 MB. Allerdings gibt es Methoden im Namensraum System.Drawing, um ein Bild in einen Thumbnail von etwa 8 KB Größe umzuwandeln. Das wird der intrinsische Zustand sein, der klein genug ist, um alle einzelnen Bilder gleichzeitig im Speicher zu halten. Über die Gruppeninformationen kann die Anwendung die Bilder in verschiedenen Kombinationen anzeigen. Und wenn sie wieder auf die Festplatte zugreift, kann sie auch die Bilder in der ursprünglichen Größe darstellen.

Design

Schauen Sie sich das UML-Diagramm in Abbildung 3-4 an. Wie im vorigen Abschnitt beschrieben, basiert das Fliegengewicht-Muster darauf, den Zustand der Anwendung in drei Typen aufteilen zu können. Der *intrinsicState* liegt in den



Abbildung 3-3: Illustration des Fliegengewicht-Musters – Foto-Gruppe

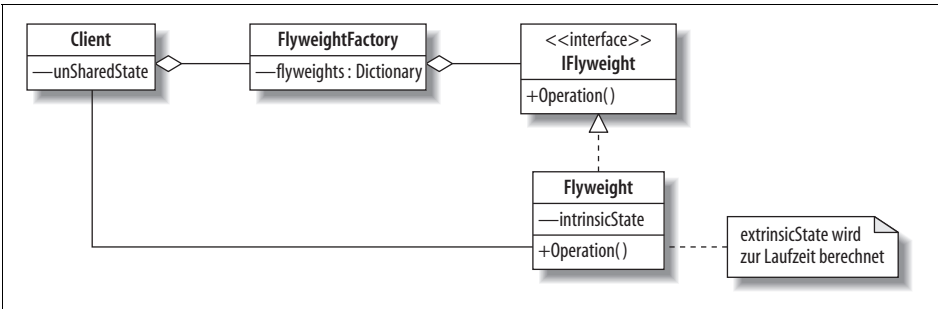


Abbildung 3-4: UML-Diagramm des Fliegengewicht-Musters

Flyweight-Objekten. Die Klasse Flyweight implementiert ein Interface IFlyweight, das die Operationen festlegt, auf denen der Rest des Systems aufbaut. Der Client verwaltet den unSharedState und ein Dictionary aller Flyweights, die er von einer FlyweightFactory erhält. Diese hat die Aufgabe, sicherzustellen, dass jeder Wert nur

einmal erstellt wird. Schließlich taucht der `extrinsicState` als solcher nicht im System auf – er soll zur Laufzeit für jede Instanz `intrinsicState` bei Bedarf errechnet werden.

Die Teilnehmer am Fliegengewicht-Muster sind:

Client

Berechnet und verwaltet den ungeteilten Zustand der Objekte.

IFlyweight

Definiert ein Interface, über das Flyweights intrinsische Zustände empfangen und mit ihnen arbeiten können.

FlyweightFactory

Erstellt und verwaltet Flyweight-Objekte.

Flyweight

Speichert den intrinsischen Zustand, der von allen Objekten genutzt werden kann.

Es gibt noch andere Design-Optionen. So wird zum Beispiel in Abbildung 3-4 Flyweight so dargestellt, dass es den `extrinsicState` berechnet. Der Client könnte aber auch die Berechnung vornehmen und den `extrinsicState` an das Flyweight als Parameter für `Operation` übergeben. Auch können wir uns vorstellen, `unSharedState` als Dictionary mit Werten zu betreiben, die mit den Flyweights verbunden sind. Auf jeden Fall kann `unSharedState` eine komplexere Struktur haben und in einer eigenen Klasse verwaltet werden. In diesem Fall würde er neben der Klasse Flyweight stehen und das Interface IFlyweight implementieren.

QUIZ

Verbinde die Teilnehmer des Fliegengewicht-Musters mit der Foto-Gruppen-Illustration

Um zu testen, ob Sie das Fliegengewicht-Muster verstanden haben, decken Sie die linke Spalte der unteren Tabelle mit der Hand ab und versuchen Sie, die Teilnehmer anhand des Beispiels (Abbildung 3-4) in der rechten Spalte zuzuordnen. Dann vergleichen Sie Ihre Antworten mit der linken Spalte.

Client xIFlyweight FlyweightFactory Flyweight intrinsicState extrinsicState unSharedState	Foto-Gruppen-Anwendung Spezifikation eines Bildes Register mit eindeutigen Bildern Erzeuger und Zeichner eines Thumbnails Thumbnail Komplettes Bild Gruppeninformationen
---	--

Implementierung

Unsere Implementierung des Fliegengewicht-Musters nutzt zwei interessante Features von C# 1.0 und drei aus C# 3.0:

- Structs
- Indexer
- Implizite Typisierung für lokale Variablen und Arrays
- Objekt- und Collection-Initialisierer
- Anonyme Typen

Zudem werden generische Collections aus C# 2.0 genutzt, die schon im Abschnitt über das Kompositum-Muster besprochen wurden. Ich werde diese Features im folgenden Text vorstellen.

Dieses Muster umfasst drei Typen: `IFlyweight`, `Flyweight` und `FlyweightFactory`. Wie im Kompositum-Muster sollten wir sie möglichst in einen Namensraum stecken und diesen Namensraum als theoretisches Beispiel für das Muster nutzen.



Da sich die Typen in einem Interface befinden, müssen sie und ihre Konstruktoren als `public` deklariert werden.

Wir beginnen mit dem Interface:

```
public interface IFlyweight {
    void Load (string filename);
    void Display (PaintEventArgs e, int row, int col);
}
```

Die Methode `Load` wird eine Thumbnail-Version eines gegebenen Bildes in den Speicher laden, wobei der Dateiname übergeben wird. Die Methode `Display` wird das Thumbnail anzeigen. Später werden wir weitere `Display`-Optionen zum Anzeigen des vollständigen Bildes (sein extrinsischer Zustand) hinzufügen müssen. Hier ist es nun Zeit, das erste unserer Sprachfeatures zu behandeln: Structs.

Ein `Flyweight` ist ein idealer Kandidat für ein Struct – `Flyweights` sind klein und erben von keinem anderen Typ (auch wenn der Typ ein Interface implementiert). Hier ist unser `Flyweight`-Struct:

```
public struct Flyweight : IFlyweight {
    // Intrinsischer Zustand
    Image pThumbnail;

    public void Load (string filename) {
        pThumbnail = new Bitmap("images/"+filename).
            GetThumbnailImage(100, 100, null, new IntPtr());
    }
}
```

```

public void Display(PaintEventArgs e, int row, int col) {
    e.Graphics.DrawImage(pThumbnail,col*100+10, row*130+40,
        pThumbnail.Width,pThumbnail.Height);
}
}

```

C#-Feature – Structs

C# hat zwei Typ-Konstrukte, die Attribute und Operationen definieren: die wohl-bekanntere *Klasse* und die weniger bekannte *Struct*. Structs sind Klassen sehr ähnlich: sie repräsentieren Typen, die Datenmember und Funktionsmember enthalten. Aber eine Variable eines Struct-Typs enthält die Daten des Struct direkt, während eine Variable eines Klassen-Typs eine Referenz auf die Daten enthält. Die Instanzen beider Arten sind *Objekte*. Aber Structs haben eine *Werte-Semantik* – das heißt, der Wert wird bei Zuweisungen übergeben, nicht die Referenz, wie dies bei Objekten geschieht, die aus Klassen instanziiert wurden.

Structs sind leichtgewichtig und ohne den Overhead implementiert. Sie sind insbesondere für kleine Datenstrukturen nützlich. Die Einschränkung von Structs ist, dass sie nicht von anderen Typen erben können. Es lassen sich aber Interfaces implementieren.

Siehe auch C# Language Specification Version 3.0, September 2007, Section 11

Wie gewünscht nutzt Load das Bild aus dem angegebenen Dateinamen, behält aber keine Kopie davon in seinem `intrinsicState`, sondern nur den Thumbnail. Die Berechnungen zum Positionieren der Thumbnails im Fenster werden in Abbildung 3-3 gezeigt. Sie können natürlich noch verbessert werden, indem man Konstanten nutzt. Beachten Sie aber trotzdem, dass `row` und `col` aus dem `unSharedState` der Gruppe abgeleitet werden, der von Client verwaltet wird.

Als nächstes wollen wir uns die Fabrik anschauen. Wir werden uns mit Fabriken in Kapitel 6 noch näher befassen, aber hier müssen Sie nur wissen, dass sie in ihrer einfachsten Form Objekte aufgrund bestimmter Bedingungen erzeugen. Wir sind an dieser Stelle daran interessiert, zu prüfen, ob ein Objekt schon existiert, bevor wir es einer Collection hinzufügen. Bevor wir uns den Code genauer anschauen, wollen wir einen kurzen Blick auf Indexer werfen.

Jetzt können wir die Klasse `FlyweightFactory` vorstellen:

```

1 public class FlyweightFactory {
2     // Hält eine indexierte Liste von IFlyweight-Objekten
3     Dictionary <string,IFlyweight> flyweights =
4         new Dictionary <string,IFlyweight> ();
5
6     public FlyweightFactory () {
7         flyweights.Clear();
8     }

```

```

9
10 public IFlyweight this[string index] {
11     get {
12         if (!flyweights.ContainsKey(index))
13             flyweights[index]=new Flyweight();
14         return flyweights[index];
15     }
16 }
17 }

```

In den Zeilen 3–4 wird ein Dictionary deklariert, das Strings auf Fliegengewichte abbildet. In den Zeilen 13–14 wird das Dictionary mit dem Indexer angesprochen, der Teil der Dictionary-Definition ist. Zeile 13 ist ein Beispiel für die Verwendung des set-Accessors und in Zeile 14 wird ein get-Accessor genutzt.

Das ist alles Teil des Member, der in den Zeilen 10–16 deklariert wird und in dem wir unseren eigenen Indexer für die Klasse FlyweightFactory definieren. Zeile 10 legt den Rückgabetyt und den Schlüsseltyp fest, während die Zeilen 12–14 den Rumpf des get-Accessors definieren. Zunächst wird hier geprüft, ob es schon einen Flyweight mit diesem Schlüssel gibt. Wenn nicht, wird ein neuer angefordert. In beiden Fällen wird das Element aus dem Dictionary mit dem korrekten Index zurückgeliefert.

C#-Feature – Indexer

Ein *Indexer* ist ein Member, der es einem Objekt ermöglicht, auf die gleiche Art und Weise über einen Index angesprochen zu werden, wie ein Array. Er wird zusammen mit Collections genutzt, so dass wir auf Collection-Objekte per Index auch über [index] anstatt über Key(index) zugreifen können. Zudem kann der Indexer-Accessor zunächst mit der Methode ContainsKey prüfen, ob der Wert existiert.

Indexer gehören zu Eigenschaften und haben die gleiche Syntax mit get und set wie Accessors. Ein Indexer wird so definiert:

```

Modifizierer Rückgabetyt this[Schlüsseltyp key] {
    get { ... }
    set { ... }
}

```

Der Inhalt des get-Blocks sollte einen Wert vom Typ Rückgabetyt liefern. Der Inhalt des set-Blocks kann den impliziten Parameter value verwenden.

Wenn ein Typ einen Indexer besitzt, kann ein Objekt dieses Typs so verwendet werden:

```

obj[index] = x;
x = obj[index];

```

Indexer sind (natürlich) schon für Arrays und für alle Collections in der .NET-Framework-Bibliothek, die von C# verwendet wird, definiert.

Siehe auch C# Language Specification Version 3.0, September 2007, Section 10.9

Wenn wir ein wenig vorausschauen, sehen wir, dass die FlyweightFactory so instanziiert wird:

```
static FlyweightFactory album = new FlyweightFactory();
```

So wird sie angesprochen:

```
album[filename].Load(filename);
```

Damit stellt der Indexer einen Collection-ähnlichen Zugriff auf ein Objekt bereit, der eine private Collection enthält.

Beispiel: Foto-Gruppe

Nun schauen Sie sich die Anwendung an, die für die Ausgabe in Abbildung 3-4 gesorgt hat. Ihr Code, vor dem sich erst noch der FlyweightPattern-Namensraum befindet, ist in Beispiel 3-4 zu sehen.



Eine Erläuterung, wie man einen Namensraum getrennt von dem Programm kompiliert, das ihn nutzt, findet sich in der Besprechung des Fassade-Musters in Kapitel 4.

Beispiel 3-4: Beispiel 3-5 Beispiel für das Fliegengewicht-Muster – Fotogruppen und der Namensraum FlyweightPattern :

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;

namespace FlyweightPattern {

    // Fliegengewicht-Muster          Judith Bishop  Sept 2007

    public interface IFlyweight {
        void Load (string filename);
        void Display (PaintEventArgs e, int row, int col);
    }

    public struct Flyweight : IFlyweight {
        // Intrinsischer Zustand
        Image pThumbnail;
        public void Load (string filename) {
            pThumbnail = new Bitmap("images/"+filename).
                GetThumbnailImage(100, 100, null, new IntPtr());
        }

        public void Display(PaintEventArgs e, int row, int col) {
            e.Graphics.DrawImage(pThumbnail,col*100+10, row*130+40,
                pThumbnail.Width,pThumbnail.Height);
        }
    }
}
```

Beispiel 3-4: Beispiel 3-5 Beispiel für das Fliegengewicht-Muster – Fotogruppen und der Namensraum FlyweightPattern (Fortsetzung):

```
public class FlyweightFactory {
    // Hält eine indexierte Liste mit IFlyweight-Objekten
    Dictionary <string,IFlyweight> flyweights =
        new Dictionary <string,IFlyweight> ();

    public FlyweightFactory () {
        flyweights.Clear();
    }

    public IFlyweight this[string index] {
        get {
            if (!flyweights.ContainsKey(index))
                flyweights[index] = new Flyweight();
            return flyweights[index];
        }
    }
}

//===== Ende des Namensraums, Beginn des Programms

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
using FlyweightPattern;

class Client {
    // gemeinsam genutzter Zustand - die Bilder
    static FlyweightFactory album = new FlyweightFactory();

    // getrennter Zustand - die Gruppen
    static Dictionary <string,List<string>> allGroups =
        new Dictionary <string,List<string>> ();

    public void LoadGroups () {
        var myGroups = new [] {
            new {Name = "Garden",
                Members = new [] {"pot.jpg", "spring.jpg",
                "barbeque.jpg", "flowers.jpg"}},
            new {Name = "Italy",
                Members = new [] {"cappuccino.jpg", "pasta.jpg",
                "restaurant.jpg", "church.jpg"}},
            new {Name = "Food",
                Members = new [] {"pasta.jpg", "veggies.jpg",
                "barbeque.jpg", "cappuccino.jpg", "lemonade.jpg" }},
            new {Name = "Friends",
                Members = new [] {"restaurant.jpg", "dinner.jpg"}}
        };
    };
}
```

Beispiel 3-4: Beispiel 3-5 Beispiel für das Fliegengewicht-Muster – Fotogruppen und der Namensraum FlyweightPattern (Fortsetzung):

```
// Lade die Flyweights u. sichere sie im gemeinsamen intrinsischen Zustand
foreach (var g in myGroups) { // implizite Typisierung
    allGroups.Add(g.Name, new List <string>());
    foreach (string filename in g.Members) {
        allGroups[g.Name].Add(filename);
        album[filename].Load(filename);
    }
}

public void DisplayGroups (Object source, PaintEventArgs e) {
    // Anzeige der Flyweights, Übergabe des getrennten Zustands
    int row;
    foreach(string g in allGroups.Keys) {
        int col;
        e.Graphics.DrawString(g,
            new Font("Arial", 16),
            new SolidBrush(Color.Black),
            new PointF(0, row*130+10));
        foreach (string filename in allGroups[g]) {
            album[filename].Display(e, row, col);
            col++;
        }
        row++;
    }
}

class Window : Form {
    Window() {
        this.Height = 600;
        this.Width = 600;
        this.Text = "Picture Groups";
        Client client = new Client();
        client.LoadGroups();
        this.Paint += new PaintEventHandler (client.DisplayGroups);
    }

    static void Main () {
        Application.Run(new Window());
    }
}
```

Die Anwendung hat zwei Phasen: zuerst werden die Gruppen geladen, dann angezeigt. Zur einfacheren Bild-Handhabung nutzen wir das Modell `Application.Run` von C#. Das `Window`-Objekt ruft explizit `LoadGroups` auf, während `DisplayGroups` implizit durch den `PaintEventHandler` gestartet wird, nachdem `Form` gezeichnet wurde. `LoadGroups` zeigt die Verwendung von impliziter Typisierung und den Initialisierungsmöglichkeiten in C# 3.0 (siehe die folgenden Kästen).

C# 3.0-Feature – Implizite Typisierung

Variablen können als Felder in Klassen oder lokal in Methoden deklariert werden. Eine lokale Variable einer Methode (aber kein Feld einer Klasse oder eines Structs) kann ihren Typ aus dem Ausdruck ableiten, durch den sie initialisiert wurde. Das wird als *implizite Typisierung* bezeichnet. Dabei wird bei der Syntax `var` an Stelle des Typs angegeben. Diese neue Syntax kann die Lesbarkeit verbessern, indem die Redundanz verringert wird. Im folgenden Beispiel wird der Typ nur einmal angegeben:

```
var marks = new Dictionary <string, int> ();
```

Die Syntax ist vor allem dann wichtig, wenn der Typ nicht benannt ist (ein *anonymer Typ*, siehe den nächsten Kasten). In diesem Fall ermöglicht es die implizite Typisierung, den Typ aus einer zugewiesenen Variablen oder einem Typ zu ermitteln, wie zum Beispiel hier:

```
foreach (var g in myGroups) { ... }
```

Siehe auch C# Language Specification Version 3.0, September 2007, Section 8.5

`myGroups` in der Methode `LoadGroups` ist ein Beispiel eines anonymen Typs mit einer Collection und einem Array-Initialisierer. Sie verwaltet die Dateinamen der Bilder in jeder Gruppe. Natürlich sollten diese Daten von der Festplatte gelesen werden, aber es ist interessant, zu sehen, wie sie im Programm über eine anonym typisierte Datenstruktur eingefügt werden können. Der Initialisierer definiert die Datenstruktur-Member (in diesem Fall `Name` und `Members`) – sie werden später in der Methode von der `foreach`-Schleife genutzt, um die Gruppen einzurichten:

```
foreach (var g in myGroups) { // implizite Typisierung
    allGroups.Add(g.Name, new List <string>());
    foreach (string filename in g.Members) {
        allGroups[g.Name].Add(filename);
        album[filename].Load(filename);
    }
}
```

Die Fliegengewicht-Aktion geschieht in der letzten Zeile, in der `album` mit einem Dateinamen angesprochen wird und entweder ein schon bestehendes gemeinsam zu nutzendes Fliegengewicht-Objekt zurückliefert oder ein neues erstellt und zurückgegeben wird. Dann wird die Methode `Load` für dieses Objekt aufgerufen, das den Typ `IFlyweight` besitzt.

Verwendung

Das Fliegengewicht-Muster kann genau dann angewendet werden, wenn alle folgenden Bedingungen zutreffen:

C# 3.0-Feature – Implizit typisierte Arrays

In C# 2.0 konnte bei Arrays nur die Anzahl der Elemente aus der einfachen Form eines Array-Initialisierers ermittelt werden. Die implizite Typisierung wurde seitdem auch auf Arrays ausgeweitet, so dass sowohl der Typ der Elemente, als auch die Anzahl aus dem Ausdruck bestimmt werden können, wie zum Beispiel hier:

```
var myGroups = new [] {  
    new {Name= "Garden",  
        Members = new [] {"pot.jpg", "spring.jpg"  
            "barbeque.jpg", "flowers.jpg"}},  
    new {Name = "Friends",  
        Members = new [] {"restaurant.jpg", "dinner.jpg"}}  
};
```

Damit würde ein neues Array mit zwei Zeilen entstehen (Zeile 0 und Zeile 1). Jedes Element ist ein Objekt mit den zwei Feldern Name und Members. Members selber ist wiederum ein Array mit unterschiedlicher Anzahl an Elementen für jeden Eintrag.

Siehe auch C# Language Specification Version 3.0, September 2007, Section 7.5.10.4

C# 3.0-Feature – Objekt- und Collection-Initialisierer

Initialisierer geben Werte für Felder oder Eigenschaften von Objekten oder Collections an. Beispiele dafür sind:

```
Point p = new Point {X = 0, Y = 1};  
List<int> digits = new List<int>  
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

In C# 2.0 war die Initialisierungssyntax nur für Arrays gültig.

Siehe auch C# Language Specification Version 3.0, September 2007, Section 7.5.10.1-3

- Es gibt eine sehr große Zahl von Objekten (Tausende), die eventuell nicht in den Arbeitsspeicher passen.
- Ein Großteil des Zustandes kann auf der Festplatte gehalten oder zur Laufzeit berechnet werden.
- Der restliche Zustand kann in eine deutlich kleinere Zahl von Objekten mit gemeinsamen Zustand zerlegt werden.

Viele Beispiele für das Fliegengewicht-Muster beziehen sich auf Textverarbeitung, wo die Flyweight-Objekte Zeichen sind. Allerdings ist das vorige Beispiel mit den Fotos heutzutage sicherlich überzeugender.

C# 3.0-Feature – Anonyme Typen

Anonyme Typen entstehen aus Objekt-Initialisierern. Ein anonymer Typ ist ein Klassentyp ohne Name, der direkt vom obersten Typ `object` erbt. Die Member eines anonymen Typs sind eine Sequenz nur lesbarer Eigenschaften, die aus dem Initialisierer für das anonyme Objekt ermittelt werden, durch den die Instanz des Typs erzeugt wurde. Indem neue Deklarationen eingebettet werden, kann ein Initialisierer Werte für eine ganze Collection aus Objekten festlegen. So würde zum Beispiel der anonyme Typ, der durch diese Deklaration erzeugt wird:

```
var group = new {Name = "Italy",  
                Members = new [] {"cappuccino.jpg", "pasta.jpg",  
                                "restaurant.jpg", "church.jpg"}},
```

die zwei Eigenschaften `Name` und `Members` besitzen, wobei `Members` ein Array aus Strings ist. Die Typen würden interne Namen haben und wären kompatibel mit jedem anderen Typ der gleichen Struktur. So können zum Beispiel `group` einem Element von `myGroups` zugewiesen werden:

```
myGroups[0] = group;
```

Automatische `get`-Eigenschaften werden für die aufgeführten Member erstellt, daher können wir auf den Wert von `group.Name` zugreifen. Es gibt keine `set`-Eigenschaft.

Siehe auch C# Language Specification Version 3.0, September 2007, Section 7.5.10.6

Verwenden Sie das Fliegengewicht-Muster, wenn ...

Sie haben:

- Viele Objekte, mit denen Sie im Arbeitsspeicher hantieren müssen
- Verschiedene Arten von Zustände, die unterschiedlich gehandhabt werden können, um Platz zu sparen
- Objekt-Gruppen, die einen Zustand gemeinsam verwenden können
- Möglichkeiten, einen Teil des Zustands zur Laufzeit zu berechnen

Sie wollen:

- Ein System trotz schwieriger Speicher-Bedingungen implementieren

Übungen

1. Implementieren Sie die Foto-Gruppen-Anwendung komplett, also auch das Laden und Anzeigen eines Bildes (extrinsischer Zustand), wenn sein Thumbnail (intrinsischer Zustand) angeklickt wurde.
2. Integrieren Sie die Foto-Bibliothek- und Foto-Gruppen-Anwendungen, so dass sowohl Befehle ausgeführt, als auch Bilder angezeigt werden können. Beobach-

ten Sie, wie die beiden Muster (Kompositum und Fliegengewicht) zusammenarbeiten und berichten Sie am Ende der Übung darüber.

3. Im weiter oben befindlichen Beispiel für das Kompositum-Muster (Beispiel 3-3) wurden die Spezifikationen für die Foto-Bibliothek aus einer Datei gelesen. Spezifizieren Sie den initialen Zustand der Bibliothek als Übung zur Objektinitialisierung im Programm und nutzen Sie dabei die gleichen Daten:

```
AddSet Home
AddPhoto Dinner.jpg
AddSet Pets      Zu einer andere Ebene wechseln
AddPhoto Dog.jpg
AddPhoto Cat.jpg
Find Album      Dafür sorgen, dass Garden auf Home-Ebene ist
AddSet Garden
AddPhoto Spring.jpg
AddPhoto Summer.jpg
AddPhoto Flowers.jpg
AddPhoto Trees.jpg
```

(Hinweis: Nehmen Sie diese Zeilen aus der Datendatei heraus. Das Programm müsste sonst nicht geändert werden, da die Schleife mit dem Verarbeiten des ersten Befehls aus der Datei beginnen wird – der dann Display ist.)

Vergleich der Muster

Ich habe das Kompositum- und das Fliegengewicht-Muster in diesem Kapitel zusammengefasst, weil sie beide mit Strukturen mehrerer Objekte arbeiten. Das Kompositum-Muster kümmert sich darum, auf Befehle für den Zugriff und die Veränderung einer Datenstruktur einheitlich zu reagieren, während das Fliegengewicht-Muster ein pfiffiger Weg ist, Arbeitsspeicher zu sparen, wenn es mehrere identische Objekte gibt. Sie lassen sich beide in einen Namensraum stecken und dann von einem Client nutzen.

In unseren Beispielen wurde der CompositePattern-Namensraum generisch programmiert, so dass jeder Client die Möglichkeit hat, das Muster seinen Anforderungen entsprechend zu instanziiieren. Die Bedingungen dazu sind, dass der Typ, der zum Instanziiieren von Component genutzt wird, die beiden Methoden Equals und ToString definiert haben muss. Die Methode Equals ist unbedingt notwendig, damit die Methode Find korrekt arbeiten kann, während auf ToString verzichtet werden kann, wenn es schwierig ist, eine sinnvolle Ausgabe zu erhalten. Der Namensraum FlyweightPattern enthält andererseits die Typen IFlyweight und Flyweight, die sehr spezifisch auf die Foto-Anwendung ausgerichtet sind. (Sie enthält zudem die Klasse FlyweightFactory, die von keiner anderen Klasse des Clients abhängt.)

Das Fliegengewicht ist ein Service-Muster – es kann für viele andere Muster genutzt werden, um die Daten kompakt zu halten. Beispiele werden wir später sehen, wenn es um die Muster Interpreter, Zustand oder Strategie geht. Das Kompositum-Mus-

ter ist ebenfalls in Kombination mit anderen Mustern nützlich, die mit Datenstrukturen umgehen müssen. Es ist dabei eher üblich, dass das Kompositum-Muster das Fliegengewicht-Muster nutzt, als umgekehrt.

Die Aufteilung von intrinsischem und extrinsischem Zustand im Fliegengewicht-Muster hat eine Parallele beim Virtuellen-Proxy-Muster. Ausgangspunkt für das Fliegengewicht-Muster war, dass der extrinsische Zustand aus dem intrinsischen Zustand berechnet werden kann. In unserem Beispiel mit der Foto-Gruppen-Anwendung berechnen wir den extrinsischen Zustand nicht so richtig, sondern nutzen nur den Dateinamen, um das Foto von der Festplatte zu laden. Das Proxy-Muster macht Ähnliches – wir können im Zweifel das komplette Element von der Festplatte laden, wenn wir es brauchen. Ein Unterschied ist, dass das Proxy-Muster mit einzelnen Objekten umgeht, während das Fliegengewicht-Muster gleich von Anfang an mit einem Objekt-Dictionary arbeitet. Die Unterschiede sind in Tabelle 3-1 zusammengefasst.

Tabelle 3-1: Vergleich der Kompositum- und Fliegengewicht-Muster

	Kompositum	Fliegengewicht
Feste Typen	IComponent <T> Component <T> Composite <T>	IFlyweight Flyweight FlyweightFactory
Generisch	Ja	FlyweightFactory ist Typ-unabhängig
Bedingungen	T implementiert Equals und ToString	Struktur und Beziehung zwischen dem intrinsischen und extrinsischen Zustand ist im Typ Flyweight eingebaut
Dazu passende Muster	Dekorierer, Erbauer, Iterator, Besucher	Kompositum, Interpreter, Zustand, Strategie, Proxy

