Aufgabe 1

Listing 1 enthält eine Bibliothek von Klassen, aus deren Instanzen man ein hierarchisch strukturiertes User Interface erzeugen kann. Die Klassen Label, Image und Group sind allesamt vom Typ UIElement abgeleitet. Mit Hilfe von Group lassen sich Listen aufbauen. Da Group selbst ein UIElement ist, können auch hierarchische Strukturen gebildet werden. Die Klasse Visitor soll es ermöglichen, Code zu schreiben, der derartige Objekt-Hierarchien traversiert und dabei für jeden im Baum vorkommenden Typ unterschiedliche Behandlungsroutinen vorhält.

a) Wie nennt man dieses Pattern?

Listing 1

```
1   public class UIElement
2   {
3       public string Name;
4       public virtual void Accept(Visitor v) {};
5   }
6
7   public class Label : UIElement
8   {
9       public string Contents;
10      public override void Accept(Visitor v) { v.Visit(this); };
11  }
12
13  public class TextInput : UIElement
14  {
15      public string Text;
16      public override void Accept(Visitor v) { v.Visit(this); };
17  }
18
19  public class Image : UIElement
20  {
21      public string Path;
22      public int Width, Height;
23      public override void Accept(Visitor v) { v.Visit(this); };
24  }
25
26  public class Group : UIElement
27  {
28      public List<UIElement> Children;
29
30      public override void Accept(Visitor v)
31      {
32          v.Visit(this)
33          foreach (UIElement uiElement in Children)
34          {
35              uiElement.Accept(v);
36          }
37      }
38  }
39
40  public class Visitor
41  {
42      public virtual void Visit(UIElement e) {}
43      public virtual void Visit(Label l)     {}
44      public virtual void Visit(TextInput t) {}
45      public virtual void Visit(Image i)     {}
46      public virtual void Visit(Group g)     {}
47  }
```

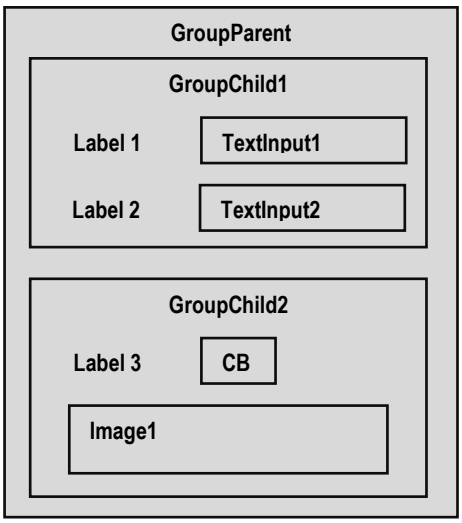
Listing 2 enthält Code, der zunächst einen eigenen Visitor-Typ implementiert und hier Code hinterlegt, der für jeden UIElement-Typ aufgerufen werden soll. In einem Hauptprogramm wird dann ein Objekt-Graph aus den UIElement-Typen erstellt und schließlich der Beispiel-Visitor darauf angewendet

b) Zeichnen Sie den einen in der Main-Methode instanziierten Objekt-Baum. Es sollen auch die aktuellen Zustände (Fields) der Objekte angegeben werden.

c) Abbildung 1 zeigt einen etwas komplexeren Dialog. Erweitern/Verändern Sie den Code in der Main Methode in Listing 2 so, dass der Dialog aus der Abbildung initialisiert wird. Bilden Sie Eltern-Kind-Verhältnisse des grafischen Aufbaus mit Hilfe des Datentyp Group ab.

d) Worin liegt der Vorteil, den Code, der Typ-spezifisch ausgeführt werden soll, in einer Klasse wie SimpleTestVisitor zu zentralisieren? Warum wurde dieser Code nicht direkt in den Accept-Methoden der Klassen Image, Label, Textinput und Group implementiert?

Abbildung 1: Ein Beispieldialog



Listing 2

```
1     public class SimpleTestVisitor : Visitor
2     {
3         public override void Visit(UIElement e)
4         {
5             Console.WriteLine("Unknown");
6         }
7         public override void Visit(Label l)
8         {
9            Console.WriteLine(l.Name " is a Label: " + l.Contents);
10        }
11        public override void Visit(TextInput t)
12        {
13           Console.WriteLine(t.Name " is a TextInput: " + t.Text);
14        }
15        public override void Visit(Image i)
16        {
17            Console.WriteLine(i.Name + " is an image. Size: (" + i.Width+ "," + i.Height + ")");
18        }
19        public override void Visit(Group g)
20        {
21            Console.WriteLine(g.Name + " is a Group with " + g.Children.Count + " children.");
22        }
23    }
24
25    public class Program
26    {
27        public static void Main(string[] args)
28        {
29            UIElement dialog =
30                new Group {Name="Ein Dialog",
31                    Children = new List<UIElement>(new UIElement[] {
32                        new Label     {Name="Label Eins", Contents="Hier steht was"},
33                        new Image     {Name="Bild Eins", Path="Smiley.jpg", Width=400, Height=300},
34                        new Textinput {Name="Input", Text="Hier was eingeben" },
35                    })};
36            SimpleTestVisitor v = new SimpleTestVisitor();
37            dialog.Accept(v);
38        }
39    }
```

Aufgabe 2

a) Stellen Sie sich vor, beim Traversieren Ihres geänderten Codes (mit dem erweiterten Dialog aus Abbildung 1) wäre ein Breakpoint in Listing 2, Zeile 17 gesetzt worden (in der Methode void Visit(Image i)). Listen Sie die Aufrufreihenfolge der Methoden auf, beginnend mit der Methode Main, die schließlich im Rumpf der Methode in Listing 2, Zeile 17 endet.

b) In Listing 1 erscheinen die Zeilen 10, 16 und 23 wie Code-Wiederholung. Erklären Sie, warum diese dennoch unverzichtbar sind, bzw. warum es nicht genügt hätte, die hier scheinbar redundante Implementierung einfach in die Implementierung der Methode Accept in der Basisklasse, also in Listing 1, Zeile 4 zu packen,

c) Was bedeutet in diesem Zusammenhang der Begriff „Double Dispatch“?

Aufgabe 3

Mit Hilfe der Möglichkeit, einen Objektgraphen zu traversieren, soll nun ein weiterer Visitor geschaffen werden, der es erlaubt, Image-Objekte mit bestimmten Eigenschaften aufzufinden. Listing 3 zeigt, wie ein solcher Visitor, hier mit dem Namen „ImageFinder“, verwendet werden kann: mit Hilfe der Methode „Find“ wird zunächst ein Bild gesucht, dessen Breite größer oder gleich 350 Pixel ist, dann ein Bild, dessen Name „Smiley“ ist. Das genaue Suchkriterium, wonach „Find“ suchen soll, wird der Methode jeweils mit einem Lambda-Ausdruck übergeben. Listing 4 enthält die (lückenhafte) Implementierung von ImageFinder.

a) Formulieren Sie den Lambda-Ausdruck „**`i => i.Width > 350`**“ in eine „normale“ (keine anonyme) Methode um und ändern Sie den Aufruf in Listing 3, Zeile 3 so ab, dass diese Methode verwendet wird.

b) Wie lauten die Typen der Parameter und des Rückgabewertes der Methode aus Aufgabe a)? **Vervollständigen Sie Listing 4 Zeile 3**.

c) Wie müsste Listing 4 Zeile 10 implementiert werden:

a) `root.Accept(this);`

oder

b) `this.Visit(root);` ?

Begründen Sie Ihre Antwort.

Listing 3

```
1   // Finde ein Bild, dessen Breite größer als 350 ist und gib dessen Namen aus
2   ImageFinder imageFinder = new ImageFinder();
3   Image found1 = imageFinder.Find(dialog, i => i.Width > 350);  /* Aufgabe 3a */
4   Console.WriteLine(found1.Name);
5
6   // Finde ein Bild mit dem Namen "Smiley" und gib dessen Breite aus
7   Image found2 = imageFinder.Find(kapitel, i => i.Name == "Smiley");
8   Console.WriteLine(found2.Width);
```

Listing 4

```
1    public class ImageFinder : Visitor
2    {
3        public delegate /* Aufgabe 3b */ FindPredicate (/* Aufgabe 3b */ i);
4        private Image _found;
5        private FindPredicate _fp;
6
7        public Image Find(UIElement root, FindPredicate fp)
8        {
9            _fp = fp; _found = null;
10           /* Aufgabe 3c */
11           return _found;
12       }
13
14       public override void Visit(Image s)
15       {
16           if (_fp(s)) _found = s;
17       }
18   }
```

Aufgabe 4

Beantworten Sie folgende Fragen.

a) Was bezeichnet der Identifizierer „FindPredicate“ in Listing 4 Zeile 3? Einen **Methodennamen** oder einen **Datentyp**?

b) Welches im Unterricht genannte Sprachmerkmal von C# wäre als Grundlage geeignet, um aus der Klasse ImageFinder einen allgemeineren „Finder“ zu machen, der nicht nur die Suche nach Image-Objekten, sondern nach beliebigen Instanzen von UIElement abgeleiteter Typen erlaubt? Als Antwort genügt **ein** Begriff!

Aufgabe 5

Die Speicherung der Kinder in einem **`Container`** soll als einfach verkettete lineare Liste implementiert werden. Folgende Implementierung wird bereits vorgeben:

Listing 3

```
1   public class Container
2   {
3
4       public class ListItem
5       {
6          public object Ob;
7          public ListItem Next;
8       }
9
10      private ListItem _firstChild;
11      private ListItem _lastChild;
12
13      public void Add(object ob)
14      {
15          ListItem li =  new ListItem{ Ob = ob };
16          if (_firstChild == null)
17              _firstChild = _lastChild = li;
18          else
19          {
20             _lastChild.Next = li;
21             _lastChild = li;
22          }
23      }
24
25      public IEnumerable<object> Children
26      {
27          get
28          {
29              /* TODO – hier fehlen 2 Zeilen Code */
30          }
31          set
32          {
33              foreach (var child in value)
34                  Add(child);
35          }
36      }
37  }
```