7 Klassen und Objekte 2

7.1 Wiederholung der grundlegenden Begriffe

In der Objektorientierten Programmierung OOP betrachtet man die Welt als eine Summe von Objekten. Zum Beispiel ist ein Haus, eine Zeichnung, ein Dreieck oder ein Punkt ein Objekt. Der objektorientierte Ansatz ist aber auch im Umgang mit Computerprogrammen ständig sichtbar. Ein Dialog in einer Windows Applikation ist ein Objekt, ein Eingabefeld ist ein Objekt und eine Befehlsschaltfläche (*button*) ist ein Objekt. Ein Objekt kann durch bestimmte Daten charakterisiert werden. Ein Haus hat z.B. einen Besitzer, eine Adresse, eine Wohnfläche, eine Gartenfläche, einen Preis usw. Ein Dreieck hat drei Seitenlängen, drei Winkel und eine Fläche und ein Punkt hat in der Ebene mindestens zwei Koordinaten. Leider verwendet man in der Literatur der OOP eine Fülle von Begriffen, die dasselbe meinen oder unter leicht unterschiedlichen Gesichtspunkten dasselbe meinen. So nennt man die Daten eines Objektes auch Attribute, Attributwerte; Datenelemente, Datenfelder oder einfach Felder (*fields*). In C# erfolgt das abfragen und setzen von Datenwerten über die Eigenschaften (*properties*) des Objektes. Ein Objekt hat immer einen bestimmten Zustand (*state*). Dieser Zustand wird durch die Attributwerte beschrieben. Ein Objekt hat aber auch ein bestimmtes Verhalten (*behavior*) seiner Umgebung gegenüber. Das Verhalten wird durch Funktionen (*functions*) oder Operationen festgelegt. Mittlerweile setzt sich dafür der Begriff Methode (*methods*) durch. Außerdem kann ein Objekt Nachrichten versenden, um die interessierte Umgebung über bestimmte Ereignisse zu informieren. Objekte müssen erzeugt werden und sie müssen eine eindeutige Identität haben. Ein existierendes Objekt belegt im Speicher Speicherplatz. Nicht mehr benötigte Objekte müssen wieder zerstört werden. In C# sorgt der so genannte *garbage collector* automatisch für die Entfernung nicht mehr benötigter Objekte aus dem Speicher.

Eine Klasse (*class*) ist eine Schablone für ein Objekt. Sie beschreibt alle Datenfelder, Eigenschaften, Methoden und Ereignisse eines Objekts. Eine Klasse ist der Bauplan, die Typdefinition nach dem ein Objekt erzeugt wird. Man nennt ein Objekt auch eine Instanz einer Klasse.

Zustand und Verhalten eines Objektes bilden eine Einheit. Ein Objekt kapselt seine Daten (seinen Zustand) und seine Methoden. Bei richtiger Implementierung können die Daten nur über die öffentlichen zugänglichen Eigenschaften und Methoden verändert werden. Ein Objekt soll die Repräsentation der Daten und auch die Details der Operationen nach außen verbergen. Damit realisiert ein gut entworfenes Objekt das Geheimnisprinzip. Die Kunst des Programmierens besteht in der Modellierung des realen Systems als zusammenhängendes System von Objekten.

7.1.1 Aufbau einer Klasse:

```
class name
{
      Konstanten
      Datenfelder, das sind die Variablen der Klasse
      Konstruktoren, Destruktoren
      Eigenschaften
      Methoden
      Ereignisse
      Indexer
}
```

Die in dieser fast vollständigen Aufzählung genannten Elemente einer Klasse werden als *class members*, eingedeutscht als Klassenmember oder einfach Member bezeichnet

7.1.2 Datenfelder

Die Datenfelder (*fields*) enthalten alle Informationen um den Zustand eines Objektes festzuhalten. Sie werden in der Regel so vereinbart, dass sie von außen nicht sichtbar sind. Wer auf ein Feld unmittelbaren Zugriff hat oder eine Methode verwenden kann wird durch die Zugriffsmodifizierter (`private`, `public`) festgelegt.

Für die elementaren Datenwerte stehen verschiedene Datentypen zur Verfügung. Die am häufigsten verwendeten Typen sind:

Datenwert	C# Typ
Ganze Zahlen	`int`
Zahlen mit Kommastellen und Exponent (3.5, 34.5E12)	`double`
Wahrheitswerte true und false	`bool`
Aufzählungen	`enum`
Zeichen	`char`
Zeichenketten	`string`

Die ersten 4 Datentypen gehören zur Familie der einfachen Datentypen. Eine Zeichenkette ist ein Objekt des Typs `System.String` und ist kein einfacher Datentyp mehr. Für die Speicherung einer Zeichenkette benötigt man ein Array für die einzelnen Zeichen. Zusätzlich steht jede Menge an Methoden zur Verfügung, um mit Zeichenketten zu arbeiten.

In der OOP bezeichnet eine Klasse auch als abstrakten Datentyp.

Statische Datenfelder

Standardmäßig werden die Datenfelder für jedes Objekt, jede Instanz einer Klasse neu erzeugt. Jede Instanz verfügt über einen eigenen Satz der in der Klasse definierten Datenfelder. Man spricht auch von *instance members*.

Mit dem Modifizierer `static` kann man ein Datenfeld auch als statisches Datenfeld definieren. Statische Datenfelder existieren nur einmal und werden erzeugt, wenn die Applikation, welche die Klasse enthält gestartet wird und sie existieren während der gesamten Laufzeit der Applikation. Diese Datenfelder kann man auch verwenden, bevor man eine Instanz der Klasse erzeugt hat.

7.1.3 Eigenschaften

Eigenschaften (*properties*) dienen zum Lesen und Setzen der Datenwerte eines Objekts. Sie haben im einfachsten Fall die Form:

```
public Datentyp Name
{
  get { return datenwert; }
  set { datenwert = value; }
}
```

Dabei kann die get- oder die set-Operation auch fehlen.

7.1.4 Konstruktor

Der Programmcode des Konstruktors wird bei der Erzeugung eines Objektes verwendet. Er hat die Form

```
public Datentyp Name(formal-parameter-list_opt )
```

Der Konstruktor dient der Initialisierung von Datenfeldern. Über Argumente kann man dem Konstruktor auch Daten mitgeben. Im Framework gibt es viele Beispiele für solche Konstruktoren. Man kann für eine Klasse auch mehrere Konstruktoren, die sich in der Parameterliste unterscheiden, definieren.

7.1.5 Methoden und Methodenaufruf

In C# ist die kleinste Einheit, die vom Compiler zur Übersetzung akzeptiert wird, eine Klasse. Methoden können nur innerhalb einer Klasse definiert werden.

Methoden werden vereinfacht nach der Regel

```
method-declaration:
  method-header method-body

method-header:
  method-modifiers_opt return-type method-name
    ( formal-parameter-list_opt )

method-body:
  block | ;
```

Die Modifizierer *(method-modifiers)* legen fest, wer die Funktion verwenden kann (`private`, `public`, …). Wird keine Angabe gemacht, so gilt die Methode als private Methode.

Return-Type ist entweder `void` oder ein Datentyp. Ist ein Rückgabewert vereinbart, so muss die Methode im Rumpf zumindest eine Anweisung

```
return expression;
```

enthalten. Das Resultat des Ausdrucks muss den vereinbarten Datentyp haben.

Beispiele:

```
(1) void PrintStatus() { … }
(2) public void Add() { … }
(3) public double Add (double a, double c) { … }
(3) public static int Parse (string s) { … }
```

(1) Eine private Methode mit dem Namen `PrintStatus`. Die Parameterliste ist leer und die Methode liefert keinen Rückgabewert.

(2) Eine öffentliche Methode mit einer leeren Parameterliste und ohne Rückgabewert.

(3) Eine öffentliche Methode mit zwei Parametern vom Typ `double` und einem Rückgabewert vom Typ `double`.

(4) Eine öffentliche statische Methode. Verlangt eine Zeichenkette als Argument und liefert einen `int` - Wert als Resultat.

Beim Aufruf, d.h. der Verwendung einer Methode muss für jeden formalen Parameter ein aktueller Parameter übergeben werden. Der Typ des aktuellen Parameters muss stimmen. Bei der Verwendung von Methoden aus dem Framework wird oft eine ganze Liste von Methoden angezeigt, die denselben Namen haben, aber eine unterschiedliche Argumentliste. Das ist in C# eine prinzipielle Eigenschaft von Methoden. Die Signatur einer Methode ergibt sich aus dem Namen (incl. Namensraum) und der Argumentliste.

Aufruf der obigen Methoden:

```
(1) PrintStatus();
(2) Add();
(3) double a = 3.5;
    double c;
    c = Add (a, 2.0);
(3) int iValue;
    string input = Console.ReadLine();
    iValue = Parse (input);
```

Vielleicht ist der Vergleich mit Funktionen in der Mathematik oder in Excel hilfreich. Eine Funktion hat immer einen Namen, eine Parameterliste und ein Resultat. In Excel werden die Parameter durch Strichpunkt getrennt, in C# durch einen Beistrich.

Will man in der Mathematik ausdrücken, dass eine Funktion f von den drei Größen x, y und z abhängig ist, schreibt man auch f(x, y, z).

In älteren Programmiersprachen (PASCAL, FORTAN) gibt es Funktionen und Prozeduren. Die Funktionen haben wie in der Mathematik immer ein unmittelbares Resultat, das ist der Rückgabewert. Prozeduren haben keinen Rückgabewert. In der Parameterliste unterscheiden

sich Funktionen und Prozeduren nicht. Die OOP und C# verwendet den allgemeineren Begriff Methode. Eine Methode kann entsprechend dem alten Sprachgebrauch eine Funktion (mit Rückgabewert) oder eine Prozedur (ohne Rückgabewert → `void`) sein.

Statische Methoden

Wichtig ist noch die Angabe `static`. Methoden dieses Typs sind Teil der Klasse und nicht des Objektes. Statische Methoden kann man auch verwenden, ohne eine Objekt erzeugt zu haben. Alle Methoden `Console.`*`MethodName`* oder die Methoden `Math.`*`MethodName`* sind statische Methoden.

Statische Methoden können nur auf statische Datenfelder zugreifen. Sie können keine instanzierte Datenfelder verwenden.

7.1.6 Erzeugung und Verwendung von Objekten

Erzeugt wird ein Objekt mit

```
ObjektTyp ObjektName = new ObjektTyp();
```

oder auch auf diese Art:

```
ObjektTyp o1, o2;
o1 =  new ObjektTyp();
o2 =  new ObjektTyp();
```

Properties werden in der Form

```
Set: ObjektName.PropertyName = expression;
Get: L-Value = ObjektName.PropertyName;
```

und Methoden in der Form

```
ObjektName.MethodenName(actual-parameterlist_opt)
```

verwendet.

Beispiel:

```
public class IntListe
{
    int [] liste;
    int n;

    public IntListe(int n)
    {
       liste = new int[n];
       n = 0;
    }

    public void Add(int newValue)
    {
       liste[n] = newValue;
       n++;
    }

    public void Remove(int i)
    {
       for (int j = i; j < n-1; j++)
          liste[j] = liste[j+1];
       n--;
    }

    public string ToText()
    {
       string s = "";
       for (int i = 0; i < n; i++)
          s += String.Format("{0}: {1}\n", i, liste[i]);
       return s;
    }

    public int Get(int i)
    {
       return liste[i];
    }

    public int Count
    {
       get { return n; }
    }
}
```

```
public class FormMain : System.Windows.Forms.Form
{
    IntListe list = new IntListe(10);
    int i = 0;  // index of current Element

    public FormMain()
    {
       InitializeComponent();
       numberUpDownIndex.Minimum = 0;
       numberUpDownIndex.Maximum = 0;
       numberUpDownIndex.Enabled = false;
    }
    …
    static void Main()
    {
       Application.Run(new FormMain());
    }

    private void numberUpDownIndex_ValueChanged(object sender,
       System.EventArgs e)
    {
       i = Convert.ToInt32(numberUpDownIndex.Value);
       editElement.Text = list.Get(i).ToString();
    }

    private void btnAdd_Click(object sender, System.EventArgs e)
    {
       int newValue = 0;
       try
       {
          newValue = int.Parse(editElement.Text);
       }
       catch
       {
          MessageBox.Show("This is not an integer number!");
          return;
       }
       list.Add(newValue);
       numberUpDownIndex.Maximum = list.Count-1;
       numberUpDownIndex.Value = (decimal) list.Count-1;
       numberUpDownIndex.Enabled = true;
    }

    private void btnDelete_Click(object sender, System.EventArgs e)
    {
       list.Remove(i);
       if (list.Count > 0)
       {
          editElement.Text = list.Get(i).ToString();
          numberUpDownIndex.Maximum = list.Count-1;
       }
       else
       {
          editElement.Text = "";
          numberUpDownIndex.Maximum = 0;
          numberUpDownIndex.Enabled = false;
       }
    }
}
```

7.2 Ereignisse

Ein Ereignis (*event*) ist ein Objekt Member, der einem Objekt oder einer Klasse das Bereitstellen von Benachrichtigungen ermöglicht. In C# ist das analog zu einem Abonnement realisiert.

Andere Objekte können sich für die Nachrichten eines Objektes als Empfänger anmelden. Das Sender Objekt verschickt die Nachricht nur an den registrierten Abonnementen und nur, wenn mindestens ein Abonnement angemeldet ist.

Um ein Ereignis zu definieren, muss zuerst ein so genannter Delegat (*delegate*) definiert werden. Ein Delegat definiert eine Signatur für eine Methode und gibt dieser Signatur einen Namen. Die Signatur legt den Rückgabewert und die Parameterliste fest. Für Events verwendet man üblicherweise folgenden Typ:

```
void nameEvent (object sender, EventArguments e)
```

`EventArguments` ist eine Klasse, die von `System.Arguments` abgeleitet (→Vererbung) werden soll. Sender ist das Objekt, das die Nachricht ausgesendet hat.

Beispiel für eine Event-Argument Klasse:

```
public class OverloadEventArgs : System.EventArgs
{
   bool overload;
   public OverloadEventArgs (bool overload)
   {
     this.overload = overload;
   }
   public bool Overload
   {
     get { return overload; }
   }
}
```

Das Delegat wird definiert als:

```
public delegate void OverloadEvent(object sender,
  OverloadEventArgs e);
```

Die Klasse definiert einen Event dieses Typs:

```
public event OverloadEvent OverloadChanged;
```

und verschickt bei Bedarf die Nachricht. Dazu wird geprüft, ob es Abonnenten gibt, dann wird ein Event Argument erzeugt und die Nachricht verschickt (*raise an event*).

```
if (OverloadChanged != null)
{
  OverloadEventArgs e = new OverloadEventArgs(newOverload);
  OverloadChanged(this, e);
}
```

Auf der Empfängerseite meldet sich ein Objekt als Empfänger an:

```
r = new Resistor();
r.OverloadChanged += new OverloadEvent(OnOverloadChanged);
```

und definiert die Methode `OnOverloadChanged` entsprechend der Signatur des Delegaten:

```
private void OnOverloadChanged(object sender,
OverloadEventArgs e)
{
if (e.Overload)
  pictureBox.Image = imageList.Images[1];
else
  pictureBox.Image = imageList.Images[0];
}
```

Will sich ein Objekt wider als Abonnement abmelden, so ist der Code dafür:

```
r.OverloadChanged -= new OverloadEvent(OnOverloadChanged);
```

7.3 Vererbung

Ein wesentliches Konzept der OOP ist die Vererbung (*inheritance*). Eine Klasse kann von einer anderen Klasse abgeleitet werden. Damit steht ein Mechanismus zur Verfügung, mit dessen Hilfe man die Fähigkeiten einer existierenden Klasse erweitern kann. Die abgeleitete Klasse erbt die Datenelemente, Methoden etc. von der Elternklasse. In C# ist dieses Konzept von Beginn an präsent, weil man in der Entwicklungsumgebung bald merkt, dass für jedes Objekt eine handvoll Methoden automatisch zur Verfügung stehen. Alle Klassen sind mindestens von der elementarsten Klasse `System.Objekt` abgeleitet.

Beim Entwurf ist es wichtig zu erkennen, wo man Vererbung nützen kann. Man findet eigentlich immer verschiedene Objekte, die einiges gemeinsamen haben, sich aber auch unterscheiden. Eine abgeleitete Klasse kann zusätzliche Datenfelder, Properties und Methoden haben, sie kann aber auch gleiche Methoden unterschiedlich implementieren. Man spricht von Spezialisierung.

Zu Recht sind geometrische Figuren ein beliebtes Beispiel um die Konzepte der Vererbung zu zeigen. Will man Vererbung verwenden, so muss man für verschiedene Objekte analysieren, was sie gemeinsamen haben und wo sie sich unterscheiden. Wir planen für die drei einfachen geometrische Figuren Kreis, Quadrat und gleichseitiges Dreieck. Die unschwer zu erkennenden gemeinsamen Eigenschaften sind eine Größe und wenn wir an die Darstellung denken, eine Farbe. Wir ergänzen das noch um einen eindeutigen Namen (*identifier* (ID)).

Wir beginnen mit einer Klasse für den Basistyp eines Wertes und nennen diesen Typ "Parameter".

```
public abstract class Figure
{
   protected double size = 1;
   protected string id;
   protected int colorNr = 0;

   public double Size
   {
      get { return size; }
      set { size = value; }
   }

   public string Id
   {
      get { return id; }
      set { id = value; }
   }

   public int ColorNr
   {
      get { return colorNr; }
      set { colorNr = value; }
   }

   public Figure()
   {
   }

}
```

Neu ist hier zunächst nur das Schlüsselwort `protected`. Das ist neben `private` und `public` ein weitere Kategorie für den Gültigkeitsbereich einer Variablen einer Property oder

einer Methode. Der Gültigkeitsbereich wird auf alle Klassen ausgedehnt, die von dieser Klasse abgeleitet sind.

Eine konkrete Figur ist ein Kreis. Die Klasse `Circle` soll die gemeinsamen Eigenschaften der Klasse `Figure` erben. Formal geschieht die Ableitung von einer anderen Klasse, man nennt sie dann die Basisklasse, mittels folgender Syntax:

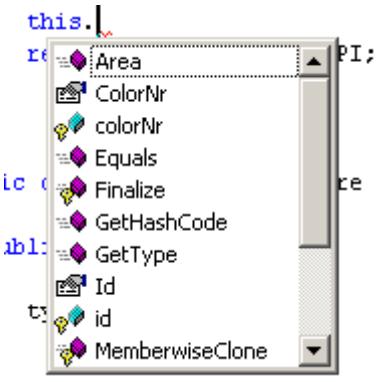
```
class class-name : base-class-name
{
}
```

Die Klasse `ParameterInt` hat ein zusätzliches Datenfeld für den eigentlichen Wert des Parameters.

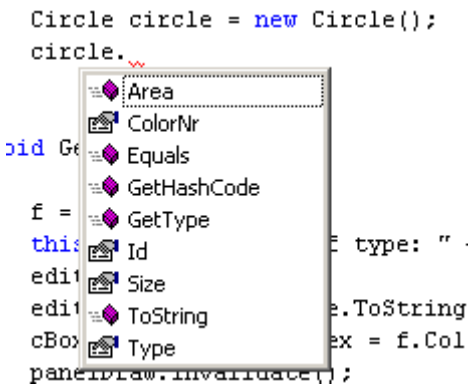
```
public class Circle : Figure
{
   public Circle()
   {
   }

   public double Area()
   {
     return size*size*Math.PI;
   }
```

Innerhalb der Klasse `Circle` stehen die Datenfelder auch für den direkten Zugriff zur Verfügung. Beachte die Symbole für die Methoden und die Datenfelder mit Schlüssel (= `protected`), mit Schloss (= `private`) und ohne zusätzliches Symbol (= `public`).

Außerhalb der Klasse sind nur noch die öffentlichen Properties und Methoden verfügbar.



Innerhalb der Klasse `Circle`



Außerhalb der Klasse `Circle`

7.3.1 Erbgut

Von der Basisklasse erbt die abgeleitete Klasse alle Member. Davon ausgenommen sind die Konstruktoren und Destruktoren. Allerdings hat die abgeleitete Klasse nur Zugriff auf jene Member, die in der Basisklasse als `protected` oder `public` vereinbart sind. Alle Datenfelder, Properties und Methoden der Basisklasse werden mit der Erzeugung eines Objekts der abgeleiteten Klasse erzeugt (instanziiert). Vererbung ist transitiv. Wenn C von B und B von A abgeleitet wird, dann erbt C sowohl von B als auch von A.

7.3.2 Virtuelle Methoden

Eine nahe liegende Eigenschaft der Vererbung ist, dass eine Methode (oder eine Eigenschaft) der abgeleiteten Klasse sich anders verhält als die gleiche Methode der Basisklasse. In unserem Beispiel mit einer Klasse `Figure` und abgeleiteten Klassen `Circle, Square, Rectangle` soll die Methode `Area` jeweils die für die Figuren richtige Fläche berechnen.

Klasse `Circle`:

```
public double Area()
{
  return size*size*Math.PI;
}
```

Klasse `Square`:

```
public double Area()
{
  return size*size;
}
```

Nun kann man diese Methoden z.B. auf diese Art verwenden:

```
Circle circle = new Circle();
circle.Size = 10.0;
double A = circle.Area();
Square square = new Square();
square.Size = 4.0;
A = square.Area();
```

Wenn wir das allerdings exakt so programmieren, nützen wir die Möglichkeiten der Vererbung, speziell der virtuellen Methoden nicht.

Virtuelle Methoden leisten folgendes:

```
Figure f1 = new Circle();
f1.Size = 10.0;
double A = f1.Area();
Figure f2 = new Square();
f2.Size = 4.0;
A = f2.Area();
```

Erzeugt werden Objekte des Typs `Circle` und `Square`. `f1` und `f2` sind Referenzen auf den Typ der Basisklasse. Trotzdem soll `f1.Area` und `f2.Area` die jeweils richtigen Methoden verwenden. Dieses Verhalten wird in C#, aber auch in C++, Java etc. durch das Konzept der virtuellen Methoden unterstützt. Der OOP Begriff für dieses Verhalten ist Polymorphismus (*polymorphism*) oder Polymorphie. Polymorphie ermöglicht es, den gleichen Namen (hier `Area`) für gleichartige Methoden zu verwenden, die auf Objekten verschiedener Klassen

auszuführen sind. Der Typ `Figure` muss nur wissen, dass es in den abgeleiteten Klassen eine Methode `Area` gibt. Erst zur Laufzeit des Programms wird bestimmt ob f1 jetzt auf einen Kreis oder ein Quadrat zeigt. Man spricht deshalb von später oder dynamischer Bindung (*late binding*).

Implementierung einer virtuellen Methode:

Die Methode wird in der Basisklasse mit dem Modifizierer `virtual` und in der abgeleiteten Klasse mit dem Modifizierer `override` versehen.

Basisklasse:

```
public virtual double Area()
{
  return 0.0;
}
```

Abgeleitete Klasse

```
public override double Area()
{
  return size*size*Math.PI;
}
```

Die Methode in der Basisklasse liefert so wie hier, oft nur ein Pseudoresultat.

Das Wort *override* wird im Wörterbuch mit "sich hinwegsetzen über, vorrangig" übersetzt. Das trifft den Sachverhalt recht gut. In einem Objekt vom Typ der abgeleiteten Klasse hat die mit dem Modifizierer "`override`" versehene Methode Vorrang vor der virtuellen Methode der Basiskasse. Im Deutschen spricht man vom Überschreiben einer Methode, eine Methode mit dem `override`-Modifizierer wird als Überschreibungsmethode bezeichnet. Eine Überschreibungsmethode überschreibt eine geerbte virtuelle Methode mit derselben Signatur.

Am meisten profitiert man von dem Konzept der virtuellen Methoden im Zusammenhang mit Auflistungen (*collections*). In einer `ArrayList` können wir z.B. eine Liste von unterschiedlichen Objekten, die entweder vom Typ der Basisklasse oder vom Typ einer davon abgeleiteten Klasse sind verwalten.

```
ArrayList list = new ArrayList();
list.Add(new Circle());// 0
list.Add(new Square());// 1
list.Add(new Triangle());   // 2
list.Add(new Circle());// 3

Debug.WriteLine("\nObjects in the list:");
foreach (Figure f in list)
{
  double A;
  f.Size = 10.0;
  A = f.Area;
  …
}
```

Eine `ArrayList` merkt sich nur object Referenzen. Eine Alternative zu obigem Code ist

```
Debug.WriteLine("\nObjects in the list:");
for (int i = 0; i < list.Count; i++)
{
  Figure f = (Figure) list[i];
  double A;
  f.Size = 10.0;
  A = f.Area;
  …
}
```

d.h. `list[i]` ist vom Typ `object` und muss daher umgewandelt (*type cast*) werden. In der `foreach` Schleife geschieht das automatisch durch den Typ Figure für die Kontrollvariable.

Als Ergänzung zu dem. was Polymorphie leistet, können wir uns überlegen, wie man dieses Verhalten angenähert nachbilden könnte. Die Basisklasse merkt sich in einem Feld den Typ des erzeugten Objekts. Dies kann man z.B. im Konstruktor der abgeleiteten Klassen sicherstellen.

```
    public abstract class Figure
{
    protected double size = 1.0;
    protected string id;
    protected string type;
    protected int colorNr = 0;

    public double Size
    {
       get { return size; }
       set { size = value; }
    }
...

    public string Type
    {
       get { return type; }
    }

    public Figure()
    {
    }

    public double Area()
    {
       switch (type)
          case "Circle":
             return size*size*Math.PI;
          case "Square":
             return size*size;
       }
    }
}

public class Square : Figure
{
    public Square()
    {
       type = "Square";
    }

}
```

```
public class Circle : Figure
{
    public Circle()
    {
       type = "Circle";
    }

}
```

Im .Net Framework sind Mechanismen eingebaut, die es ermöglichen, zur Laufzeit jede Menge von Informationen zu einem Objekt zu ermitteln (Reflection). Jedes Objekt hat zudem die Methode `GetType` verfügbar.

```
f = (Figure) list[i];
this.Text = "Object of type: " + f.GetType().Name;
```

Eine Methode kann auf die überschriebene Basismethode zugreifen. Dazu ist ein *base-access* notwendig. Ähnlich wie `this` auf die aktuelle Klasse verweist, verweist `base` auf die Basisklasse. Der Aufruf der Basisklassenmethode innerhalb einer abgeleiteten Klasse ist sogar häufig und erfolgt in der Regel entweder am Beginn oder am Ende der Überschreibungsmethode.

Beispiel:

```
class A
{
    int x;
    public virtual void PrintFields()
   {
      Console.WriteLine("x = {0}", x);
   }
}
class B: A
{
   int y;
   public override void PrintFields()
   {
     base.PrintFields();
     Console.WriteLine("y = {0}", y);
   }
}
```

7.3.3 Verdeckte Member

Wenn in einer abgeleiteten Klasse Member mit demselben Namen oder derselben Signatur deklariert werden, so werden die entsprechenden geerbten Member verdeckt. Verdeckte Member der Basisklasse können in der abgeleiteten Klasse nicht verwendet werden. Der Compiler erzeugt eine Fehlermeldung, die mit dem Modifizierer `new` unterdrückt werden kann. Diesen Modifizierer sollte man grundsätzlich verwenden, wenn man ein Member der Basisklasse in einer abgeleiteten Klasse neu deklarieren will.

Das folgende Beispiel zeigt den Unterschied zwischen überschriebenen virtuellen Methoden und verdeckten Methoden.

```
public class BaseClass
{
    public BaseClass()
    {}

    public void BaseClassMethod()
    {
       Console.WriteLine("  Method declared in the base class\n");
    }

    public virtual void Info1()
    {
       Console.WriteLine("  Info1: Base class says hello!\n");
    }

    public void Info2()
    {
       Console.WriteLine("  Info2: Base class says hello!\n");
    }
}


public class DerivedClass : BaseClass
{

    public DerivedClass()
    {}

    public void DerivedClassMethod()
    {
       Console.WriteLine("  Method declared in the derived class!\n");
    }

    public override void Info1()
    {
       Console.WriteLine("  Info1: Derived class says hello!\n");
    }

    public new void Info2()
    {
       Console.WriteLine("  Info2: Derived class says hello!\n");
    }

}
```

Die Programmzeilen

```
BaseClass b1 = new BaseClass();
BaseClass b2 = new DerivedClass();
DerivedClass b3 = new DerivedClass();
b1.Info1();
b2.Info1();
b3.Info1();
b1.Info2();
b2.Info2();
b3.Info2();
```

erzeugen dann folgende Ausgaben:

```
Info1: Base class says hello!
Info1: Derived class says hello!
Info1: Derived class says hello!
Info2: Base class says hello!
Info2: Base class says hello!
Info2: Derived class says hello!
```

7.3.4 Versiegelte Methoden und Klassen

Man kann die Möglichkeit, virtuelle Methoden zu überschreiben explizit verhindern. Wird eine Methode zusätzlich zum Modifizierer `override` mit dem Modifizierer `sealed` versehen, so kann diese Methode in einer weiteren abgeleiteten Klasse nicht mehr überschrieben werden.

Beispiel:

```
class A
{
  public virtual void M1()
  {
    Console.WriteLine("A.M1");
  }
  public virtual void M2()
  {
    Console.WriteLine("A.M2");
  }
}

class B : A
{
  public override void M1()
  {
    Console.WriteLine("B.M1");
  }
  sealed override public void M2()
  {
    Console.WriteLine("B.M2");
  }
}
```

```
class C : B
{
  override public void M1()
  {
    Console.WriteLine("C.M1");
  }

  override public void M2()
  {
    Console.WriteLine("C.M2");
  }
}
```

Der Versuch, die Methode M2 in der Klasse C zu überschreiben, führt auf eine Fehlermeldung des Compilers. "*Cannot override inherited member B.M2() because it is sealed.*"

Auch eine Klasse kann als versiegelt deklariert werden. Von dieser Klasse kann dann nicht abgeleitet werden.

7.3.5 Abstrakte Methoden und Klassen

Eine abstrakte Methode deklariert eine neue virtuelle Methode, stellt jedoch keine Implementierung dieser Methode zur Verfügung. Der Rumpf der Methode fehlt daher, bzw. besteht nur aus einem Strichpunkt. Damit erzwingt man ein überschreiben dieser Methode in den abgeleiteten Klassen. Abstrakt wird eine Methode durch den Modifizierer `abstract`. Da eine abstrakte Methode implizit eine virtuelle Methode ist, ersetzt der Modifizierer `abstract` den Modifizierer `virtual` und kann nicht mit ihm kombiniert werden.

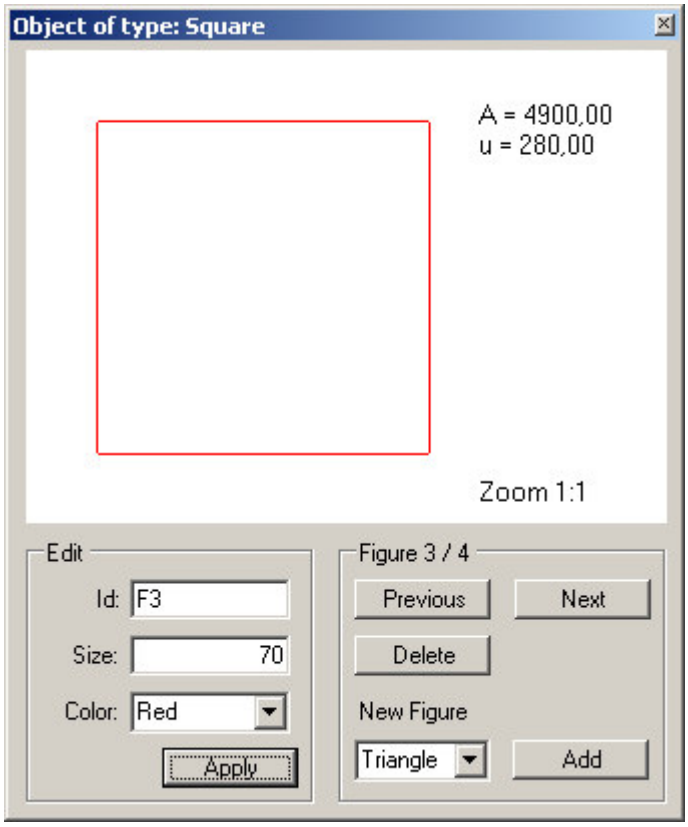
Abstrakte Methoden sind nur in abstrakten Klassen zulässig. Eine abstrakte Klasse kann nicht instanziiert werden und macht daher nur Sinn, wenn sie als Basisklasse für abgeleitete Klassen verwendet wird.

Die Basisklasse `Figure` ist ein typischer Kandidat für eine abstrakte Klasse, weil Objekte dieses Typs keinen Sinn machen.

7.3.6 Beispiel: Zeichnen der Figuren

Zur Abrundung noch eine virtuelle Methode Draw, welche die Figuren zeichnet. Das ein Objekt sich zeichnen kann ist ein typischer objektorientierter Ansatz. Allerdings braucht die Methode Informationen zur Zeichenfläche und zu den Zeichenwerkzeugen. Diese Informationen enthält der so genannte Graphikkontext.

Das Bild zeigt eine Benutzerschnittstelle zur Verwaltung und Darstellung einer Liste von geometrischen Figuren.



Das Zeichnen der Figur in einem Panel ist folgendermaßen implementiert.

```
public abstract class Figure
{
    protected float size = 30.0f;
    protected string id;
    protected int colorNr = 0;

    // drawing helpers
    protected readonly Color[] colors = { Color.Red, Color.Blue, Color.Green };
    protected Graphics grfx;
    protected float xPageMax, yPageMax;
    protected Pen pen;

...

    public Figure()
    {
    }

    public virtual double Area()
    {
       return 0.0;
    }

    public virtual double Circumference()
    {
       return 0.0;
    }
```

```
      public virtual void Draw (Control control)
    {
       grfx = control.CreateGraphics();
       grfx.PageUnit = GraphicsUnit.Pixel;

       float yDevMax = grfx.VisibleClipBounds.Height;  // in pixel
       float pageScale;
       grfx.PageUnit = GraphicsUnit.Millimeter;
       // the hight of the panel corresponds to 100 mm ( = yPageMax )
       yPageMax = 100.0f;
       // transformation from page to device coordinates:
       // yDev = yPage x PageScale x DpiY / 25.4
       // yDevMax = yPageMax x PageScale x DpiY / 25.4
       // ==>
       // PageScale = yDevMax / yPageMax / DpiY * 25.4
       pageScale = yDevMax / yPageMax / grfx.DpiY * 25.4f;
       // for sizes > 100 or < 10 we calculate a zoom factor
       // excercise: Try to understand the formulas in the next two lines
       double exp = Math.Floor(Math.Log10(size / yPageMax)) + 1;
       float zoom = (float) Math.Pow (10.0, exp);
       grfx.PageScale = pageScale/zoom;
       xPageMax = grfx.VisibleClipBounds.Width;
       yPageMax = grfx.VisibleClipBounds.Height;
       Font font = new Font(control.Font.FontFamily, 10.0f);

       string s = String.Format("A = {0:f2}\nu = {1:f2}",
          Area(), Circumference());
       grfx.DrawString(s, font, Brushes.Black, 0.7f*xPageMax, 0.1f*yPageMax);
       if (zoom >= 1.0f)
          s = String.Format("Zoom 1:{0:f0}", zoom);
       else
          s = String.Format("Zoom {0:f0}:1", 1.0f/zoom);

       grfx.DrawString(s, font, Brushes.Black, 0.7f*xPageMax, 0.9f*yPageMax);

       pen = new Pen(Color.Black , yPageMax/200.0f);

    }
}

////////////////////////////////////////////////////////////////////////

public class Square : Figure
{
    public Square()
    {
    }

    public override double Area()
    {
       return size*size;
    }

    public override double Circumference()
    {
       return 4*size;
    }

    public override void Draw(Control control)
    {
       base.Draw(control);
       pen.Color= colors[colorNr];
       grfx.DrawRectangle(pen, yPageMax/2-size/2, yPageMax/2-size/2, size, size);
       }
    }
```

```
public virtual void Draw (Control control)
{
  grfx = control.CreateGraphics();
  grfx.PageUnit = GraphicsUnit.Pixel;
...
```

Was man zum Zeichnen braucht wird in einem Objekt des Typs Graphics zur Verfügung gestellt. Alle auf einer Form verwendeten Steuerelemente sind von der Basisklasse `Control` abgeleitet und erben die Methode `CreateGraphics`. `CreateGraphics` erzeugt ein `Graphics`-Objekt.

Die grundsätzliche Anforderung an ein Graphik-Objekt ist die Möglichkeit geräteunabhängigen Code schreiben zu können. Damit ergibt sich automatisch, dass geräteunabhängige Koordinatensysteme und Koordinatentransformationen zur Verfügung stehen. Die Umrechnung der verwendeten Koordinaten in Gerätekoordinaten soll automatisch erfolgen. Für den Bildschirm (also für die Steuerelemente auf einer Form) sind zunächst Pixel als Einheit eingestellt. Viele Eigenschaften und Methoden liefern die Informationen zur aktuellen Zeichenfläche in der aktuell eingestellten Einheit.

```
grfx.PageUnit = GraphicsUnit.Pixel;    // is default
float yDevMax = grfx.VisibleClipBounds.Height;  // in pixel
```

yDevMax ist die Höhe der zur Verfügung stehenden Zeichenfläche in Pixel. Jetzt wählen wir mm als Einheit. (Am Bildschirm ist das dann nicht automatisch maßstabsgetreu!)

```
grfx.PageUnit = GraphicsUnit.Millimeter;
```

In den Zeichenoperationen werden dann alle Koordinaten mit der folgenden Formel in Pixel am Bildschirm umgerechnet:

```
yDev = yPage x PageScale x DpiY / 25.4
```

`yPage` ist ein y-Wert in den aktuellen Einheiten. `PageScale` ist ein zusätzlicher Skalierungsfaktor. `DpiY` sind *Dots* per *Inch* in y-Richtung. x-Werte werden aber immer noch von links nach rechts und y-Werte von oben nach unten größer. Hinweis: Aber auch das kann man ändern.

Der restliche Code berechnet dann `PageScale` so, dass die Höhe des Panels 100 Einheiten, also 100 mm entspricht. Ein zusätzlicher Zoomfaktor 10^n kommt zum Einsatz, wenn die Größe der Figur < 10 oder >= 100 ist.

```
xPageMax = grfx.VisibleClipBounds.Width;
yPageMax = grfx.VisibleClipBounds.Height;
```

berechnet dann Breite und Höhe der Zeichenfläche in den aktuellen Einheiten. Aufpassen muss man noch bei der Größe des Stifts. Im Konstruktor

```
pen = new Pen(Color.Black , yPageMax/200.0f);
```

ist das zweite Argument die Breite des Stifts in aktuellen Koordinaten.

Die Draw-Methode der Basisklasse erledigt die gemeinsamen Aufgaben und wird in den überschriebenen Methoden zuerst aufgerufen. Das Graphics Objekt sowie einige andere Variablen müssen den Aufruf der Basisklasse überleben und sind deshalb außerhalb der Methode als Hilfsvariablen deklariert.