clean code developer

Eine Übung, bei der Sie nur gewinnen konnten

Vier gewinnt. Eine Lösung.

Die Aufgabe war, das Spiel „Vier gewinnt" zu implementieren. Auf den ersten Blick ist das eine eher leichte Übung. Erst bei genauerem Hinsehen erkennt man die Schwierigkeiten. Wie zerlegt man beispielsweise die Aufgabenstellung, um überschaubare Codeeinheiten zu erhalten?

Leser, die sich der Aufgabe angenommen haben, ein *Vier-gewinnt*-Spiel zu implementieren [1], werden es gemerkt haben: Der Teufel steckt im Detail. Der Umgang mit dem Spielfeld, das Erkennen von Vierergruppen, wo soll man nur anfangen? Wer zu früh gezuckt hat und sofort mit der Codeeingabe begonnen hat, wird es vielleicht gemerkt haben: Die Aufgabe läuft aus dem Ruder, wächst einem über den Kopf.

Das ging mir nicht anders. Früher. Heute setze ich mich erst mit einem Blatt Papier hin, bevor ich beginne, Code zu schreiben. Denn die erste Herausforderung besteht nicht darin, das Problem zu lösen, sondern es zu verstehen.

Beim *Vier-gewinnt*-Spiel war eine Anforderung bewusst ausgeklammert: die Benutzerschnittstelle. In der Aufgabe geht es um die Logik des Spiels. Am Ende soll demnach eine Assembly entstehen, in der die Spiellogik enthalten ist. Diese kann dann in einer beliebigen Benutzerschnittstelle verwendet werden.

Beim Spiel selbst hilft es, sich die Regeln vor Augen zu führen. Zwei Spieler legen abwechselnd gelbe und rote Spielsteine in ein 7 x 6 Felder großes Spielfeld. Derjenige, der als Erster vier Steine seiner Farbe nebeneinander liegen hat, hat das Spiel gewonnen. Hier hilft es, sich mögliche Vierergruppen aufzumalen, um zu erkennen, welche Konstellationen im Spielfeld auftreten können.

Nachdem ich das Problem durchdrungen habe, zeichnet sich eine algorithmische Lösung ab. Erst jetzt beginne ich, die gesamte Aufgabenstellung in Funktionseinheiten zu zerlegen. Ich lasse zu diesem Zeitpunkt ganz bewusst offen, ob eine Funktionseinheit am Ende eine Methode, Klasse oder Komponente ist. Wichtig ist erst einmal, dass jede Funktionseinheit eine klar definierte Aufgabe hat.

Hat sie mehr als eine Aufgabe, zerlege ich sie in mehrere Funktionseinheiten. Stellt man sich die Funktionseinheiten als Baum vor, in dem die Abhängigkeiten die verschiedenen Einheiten verbinden, dann steht auf oberster Ebene das gesamte Spiel. Es zerfällt in weitere Funktionseinheiten, die eine Ebene tiefer angesiedelt sind. Diese können wiederum zerlegt werden. Bei der Zerlegung können zwei unterschiedliche Fälle betrachtet werden:

- vertikale Zerlegung,
- horizontale Zerlegung.

Der Wurzelknoten des Baums ist das gesamte Spiel. Diese Funktionseinheit ist jedoch zu komplex, um sie „in einem Rutsch" zu implementieren. Also wird sie zerlegt. Durch die Zerlegung entsteht eine weitere Ebene im Baum. Dieses Vorgehen bezeichne ich daher als vertikale Zerlegung.

Kümmert sich eine Funktionseinheit um mehr als eine Sache, wird sie horizontal zerlegt. Wäre es beispielsweise möglich, einen Spielzustand in eine Datei zu speichern, könnte das Speichern im ersten Schritt in der Funktionseinheit Spiellogik angesiedelt sein. Dann stellt man jedoch fest, dass diese Funktionseinheit für mehr als eine Verantwortlichkeit zuständig wäre, und zieht das Speichern heraus in eine eigene Funktionseinheit. Dies bezeichne ich als horizontale Zerlegung.

Erst wenn die Funktionseinheiten hinreichend klein sind, kann ich mir Gedanken darum machen, wie ich sie implementiere. Im Falle des *Vier-gewinnt*-Spiels zerfällt das Problem in die eigentliche Spiellogik und die Benutzerschnittstelle. Die Benutzerschnittstelle muss in diesem Fall nicht weiter zerlegt werden. Das mag in komplexen Anwendungen auch mal anders sein. Diese erste Zerlegung der Gesamtaufgabe zeigt Abbildung 1.

Die Spiellogik ist mir als Problem noch zu groß, daher zerlege ich diese Funktionseinheit weiter. Dies ist eine vertikale Zerlegung, es entsteht eine weitere Ebene im Baum. Die Spiellogik zerfällt in die Spielregeln und den aktuellen Zustand des Spiels. Die Zerlegung ist in Abbildung 2 dargestellt. Die Spielregeln sagen zum Beispiel aus, wer das Spiel beginnt, wer den nächsten Zug machen darf et cetera.

Der Zustand des Spiels wird beim echten Spiel durch das Spielfeld abgebildet. Darin liegen die schon gespielten Steine. Aus dem Spielfeld geht jedoch nicht hervor, wer als Nächster am Zug ist. Für die Einhaltung der Spielregeln sind beim echten Spiel die beiden Spieler verantwortlich, in meiner Implementierung ist es die Funktionseinheit *Spielregeln*.

Ein weiterer Aspekt des Spielzustands ist die Frage, ob bereits vier Steine den Regeln entsprechend zusammen liegen, sodass ein Spieler gewonnen hat. Ferner birgt der Spielzustand das Problem, wohin der nächste gelegte Stein fällt. Dabei bestimmt der Spieler die Spalte und der Zustand des Spielbretts die Zeile: Liegen bereits Steine in der Spalte, wird der neue Spielstein zuoberst auf die schon vorhandenen gelegt.

Damit unterteilt sich die Problematik des Spielzustands in die drei Teilaspekte

- Steine legen,
- nachhalten, wo bereits Steine liegen,
- erkennen, ob vier Steine zusammen liegen.

Vom Problem zur Lösung

Nun wollen Sie sicher so langsam auch mal Code sehen. Doch vorher muss noch geklärt werden, was aus den einzelnen Funktionseinheiten werden soll. Werden sie jeweils eine Klasse? Eher nicht, denn dann wären Spiellogik und Benutzerschnittstelle nicht ausreichend getrennt. Somit werden Benutzerschnittstelle und Spiellogik mindestens eigenständige Komponenten. Die Funktionseinheiten innerhalb der Spiellogik hängen sehr eng zusammen. Alle leisten einen Beitrag zur Logik. Ferner scheint mir die Spiellogik auch nicht komplex genug, um sie weiter aufzuteilen. Es bleibt also bei den beiden Komponenten Benutzerschnittstelle und Spiellogik.

Um beide zu einem lauffähigen Programm zusammenzusetzen, brauchen wir noch ein weiteres Projekt. Seine Aufgabe ist es, eine EXE-Datei zu erstellen, in der die beiden

Komponenten zusammengeführt werden. So entstehen am Ende drei Komponenten.

Abbildung 3 zeigt die Solution für die Spiellogik. Sie enthält zwei Projekte: eines für die Tests, ein weiteres für die Implementierung.

Die Funktionseinheit *Spielzustand* zerfällt in drei Teile. Beginnen wir mit dem Legen von Steinen. Beim Legen eines Steins in das Spielfeld wird die Spalte angegeben, in die der Stein gelegt werden soll. Dabei sind drei Fälle zu unterscheiden: Die Spalte ist leer, enthält schon Steine oder ist bereits voll.

Es ist naheliegend, das Spielfeld als zweidimensionales Array zu modellieren. Jede Zelle des Arrays gibt an, ob dort ein gelber, ein roter oder gar kein Stein liegt. Der erste Index des Arrays bezeichnet dabei die Spalte, der zweite die Zeile. Beim Platzieren eines Steins muss also der höchste Zeilenindex innerhalb der Spalte ermittelt werden. Ist dabei das Maximum noch nicht erreicht, kann der Stein platziert werden.

Bleibt noch eine Frage: Wie ist damit umzugehen, wenn ein Spieler versucht, einen Stein in eine bereits gefüllte Spalte zu legen? Eine Möglichkeit wäre: Sie stellen eine Methode bereit, die vor dem Platzieren eines Steins aufgerufen werden kann, um zu ermitteln, ob dies in der betreffenden Spalte möglich ist. Der Code sähe dann ungefähr so aus:

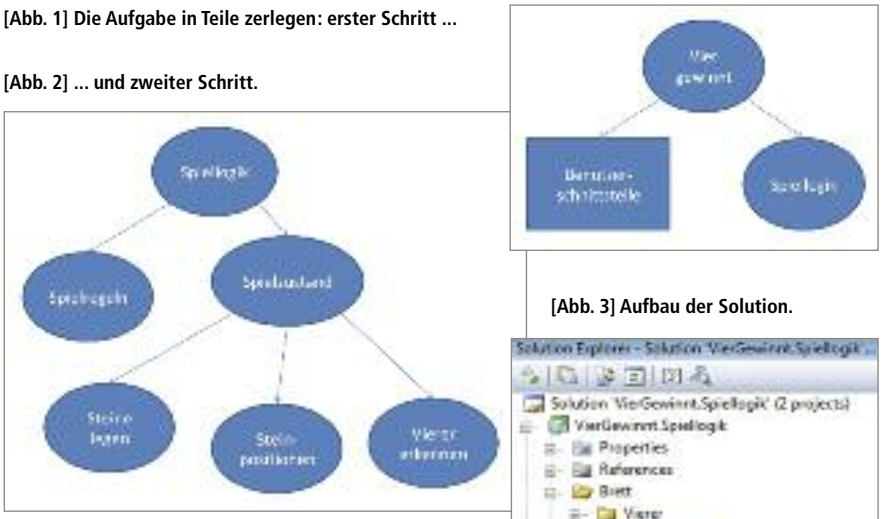
```
if(spiel.KannPlatzieren(3)) {
   spiel.LegeSteinInSpalte(3);
}
```

Dabei gibt der Parameter den Index der Spalte an, in die der Stein platziert werden soll. Das Problem mit diesem Code ist, dass er gegen das Prinzip „Tell don't ask" verstößt. Als Verwender der Funktionseinheit, die das Spielbrett realisiert, bin ich gezwungen, das API korrekt zu bedienen. Bevor ein Spielstein mit *LegeSteinInSpalte()* in das Spielbrett gelegt wird, müsste mit *KannPlatzieren()* geprüft werden, ob dies überhaupt möglich ist. Nach dem „Tell don't ask"-Prinzip sollte man Klassen so erstellen, dass man den Objekten der Klasse mitteilt, was zu tun ist – statt vorher nachfragen zu müssen, ob man eine bestimmte Methode aufrufen darf. Im Übrigen bleibt bei der Methode *LegeSteinInSpalte()* das Problem bestehen: Was soll passieren, wenn die Spalte bereits voll ist?

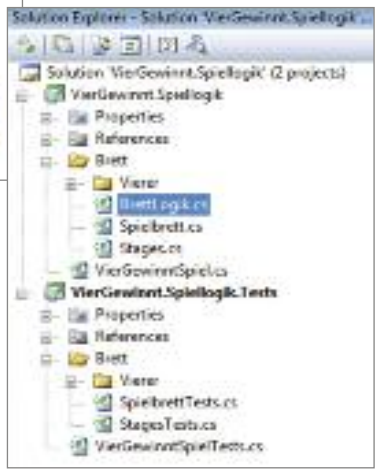
Eine andere Variante könnte sein, die Methode *LegeSteinInSpalte()* mit einem Rückgabewert auszustatten. War das Platzieren erfolgreich, wird *true* geliefert, ist die Spalte bereits voll, wird *false* geliefert. In dem Fall müsste sich der Verwender der Methode mit dem Rückgabewert befassen. Am Ende soll der Versuch, einen Stein in eine bereits gefüllte Spalte zu platzieren, dem Benutzer gemeldet werden. Also müsste der Rückgabewert bis in die Benutzerschnittstelle transportiert werden, um dort beispielsweise eine Messagebox anzuzeigen.

[Abb. 1] Die Aufgabe in Teile zerlegen: erster Schritt ...

[Abb. 2] ... und zweiter Schritt.



[Abb. 3] Aufbau der Solution.



Die Idee, die Methode mit einem Rückgabewert auszustatten, verstößt jedoch ebenfalls gegen ein Prinzip, nämlich die „Command/Query Separation". Dieses Prinzip besagt, dass eine Methode entweder ein Command oder eine Query sein sollte, aber nicht beides. Dabei ist ein Command eine Methode, die den Zustand des Objekts verändert. Für die Methode *LegeSteinInSpalte()* trifft dies zu: Der Zustand des Spielbretts ändert sich dadurch. Eine Query ist dagegen eine Methode, die eine Abfrage über den Zustand des Objekts enthält und dabei den Zustand nicht verändert. Würde die Methode *LegeSteinInSpalte()* einen Rückgabewert haben, wäre sie dadurch gleichzeitig eine Query.

Nach diesen Überlegungen bleibt nur eine Variante übrig: Die Methode *LegeSteinInSpalte()* sollte eine Ausnahme auslösen, wenn das Platzieren nicht möglich ist. Die Ausnahme kann in der Benutzerschnittstelle abgefangen und dort in einer entsprechenden Meldung angezeigt werden. Damit entfällt die Notwendigkeit, einen Rückgabewert aus der Spiellogik bis in die Benutzerschnittstelle zu transportieren. Ferner sind die Prinzipien „Tell don't ask" und „Command/Query Separation" eingehalten.

Vier Steine finden

Nun sind mit dem zweidimensionalen Array und der Methode *LegeSteinInSpalte()* bereits zwei Teilprobleme des Spielzustands gelöst: Im zweidimensionalen Array ist der Zustand des Spielbretts hinterlegt, und die Methode *LegeSteinInSpalte()* realisiert die Platzierungslogik. Das dritte Problem ist die Erkennung von Vierergruppen, also eines Gewinners.

Vier zusammenhängende Steine können beim *Vier-gewinnt*-Spiel in vier Varianten auftreten: horizontal, vertikal, diagonal nach oben, diagonal nach unten.

Diese vier Varianten gilt es zu implementieren. Dabei ist wichtig zu beachten, dass die vier Steine unmittelbar zusammen liegen müssen, es darf sich also kein gegnerischer Stein dazwischen befinden.

Ich habe zuerst versucht, diese Vierergruppenerkennung direkt auf dem zweidimensionalen Array zu lösen. Dabei habe ich festgestellt, dass das Problem in zwei Teilprobleme zerlegt werden kann:

- Ermitteln der Indizes benachbarter Felder.
- Prüfung, ob vier benachbarte Felder mit Steinen gleicher Farbe besetzt sind.

Für das Ermitteln der Indizes habe ich daher jeweils eigene Klassen implementiert, welche die Logik der benachbarten Indizes enthalten. Eine solche Vierergruppe wird mit einem Startindex instanziert und liefert dann die Indizes der vier benachbarten Felder. Diese Vierergruppen werden anschließend verwendet, um im Spielfeld zu ermitteln, ob die betreffenden Felder alle

Listing 1

Vierergruppe ermitteln.

```
internal struct HorizontalerVierer : IVierer
{
   private readonly int x;
   private readonly int y;
   public HorizontalerVierer(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public Koordinate Eins {
      get { return new Koordinate(x, y); }
   }
   public Koordinate Zwei {
      get { return new Koordinate(x + 1, y); }
   }
   public Koordinate Drei {
      get { return new Koordinate(x + 2, y); }
   }
   public Koordinate Vier {
      get { return new Koordinate(x + 3, y); }
   }
   public override string ToString() {
      return string.Format("Horizontal X: {0},
                           Y: {1}", x, y);
   }
}
```

Steine derselben Farbe enthalten. Die betreffenden Klassen heißen *HorizontalerVierer*, *VertikalerVierer*, *DiagonalHochVierer* und *DiagonalRunterVierer*. Listing 1 zeigt exemplarisch die Klasse *HorizontalerVierer*.

Zunächst fällt auf, dass die Klasse *internal* ist. Sie wird im Rahmen der Spiellogik nur intern benötigt, daher soll sie nicht außerhalb der Komponente sichtbar sein. Damit Unit-Tests für die Klasse möglich sind, habe ich auf der Assembly das Attribut *InternalsVisibleTo* gesetzt. Dadurch kann die Assembly, welche die Tests enthält, auf die internen Details zugreifen.

Aufgabe der Klasse *HorizontalerVierer* ist es, vier Koordinaten zu horizontal nebeneinander liegenden Spielfeldern zu liefern. Dies erfolgt in den Properties *Eins*, *Zwei*, *Drei* und *Vier*. Dort werden jeweils die Indizes ermittelt.

Das Ermitteln eines Gewinners geschieht anschließend in einem Flow aus zwei Schritten. Im ersten Schritt wird aus einem Spielfeld die Liste der möglichen Vierergruppen bestimmt. Im zweiten Schritt wird aus dem Spielfeld und den möglichen Vierergruppen ermittelt, ob eine der Vierergruppen Steine derselben Farbe enthält.

Die beiden Schritte des Flows sind als Extension Methods realisiert. Dadurch sind sie leicht isoliert zu testen. Anschließend können sie hintereinander ausgeführt, also als Flow zusammengeschaltet werden:

```
var gewinnerVierer = spielfeld
   .AlleVierer()
   .SelbeFarbe(spielfeld);
```

Der Flow wird an zwei Stellen verwendet: zum einen beim Ermitteln des Gewinners, zum anderen, um zu bestimmen, welche Steine zum Sieg geführt haben. Da die Methode *AlleVierer()* ein *IEnumerable* liefert und *SelbeFarbe()* dies als ersten Parameter erwartet, können die beiden Extension Methods hintereinander geschrieben werden. Da das Spielfeld in beiden Methoden benötigt wird, verfügt *SelbeFarbe()* über zwei Parameter.

Das Ermitteln von vier jeweils nebeneinander liegenden Feldern übernimmt die Methode *AlleVierer()*. Ein kurzer Ausschnitt zeigt die Arbeitsweise:

```
internal static IEnumerable<IVierer>
AlleVierer(this int[,] feld) {
    for (var x = 0; x <=
        feld.GetLength(0) - 4; x++) {
        for (var y = 0; y <
            feld.GetLength(1); y++) {
            yield return new
            HorizontalerVierer(x, y);
        }
    }
    // Ebenso für Vertikal und Diagonal
}
```

Auch diese Methode ist *internal*, da sie außerhalb der Komponente nicht benötigt wird. In zwei geschachtelten Schleifen werden die Anfangsindizes von horizontalen Vierergruppen ermittelt. Für jeden Anfangsindex wird mit *yield return* eine Instanz eines *HorizontalerVierer*s geliefert. Dieser übernimmt das Ermitteln der drei anderen Indizes.

Eine Alternative zur gezeigten Methode wäre, die möglichen Vierer als Konstanten zu hinterlegen. Es würde dann die Berechnung in *AlleVierer()* entfallen, ferner die Klassen *HorizontalerVierer* et cetera. Ob die Felder einer Vierergruppe alle mit Steinen der gleichen Farbe besetzt sind, analysiert die Methode *SelbeFarbe()*. Durch die Verwendung der Klassen *HorizontalerVierer* et cetera ist dies einfach: Jeder Vierer liefert seine vier Koordinaten. Damit muss nur noch im Spielfeld nachgesehen werden, ob sich an allen vier Koordinaten Steine gleicher Farbe befinden, siehe Listing 2.

Am Ende müssen die einzelnen Funktionseinheiten nur noch gemeinsam verwendet werden. Die dafür verantwortliche Klasse heißt *VierGewinntSpiel*. Sie ist *public* und repräsentiert nach außen die Komponente. Die Klasse ist für die Spielregeln zuständig. Da das abwechselnde Ziehen so einfach ist, habe ich mich entschlossen, diese Logik nicht auszulagern. In der Methode *LegeSteinInSpalte(int spalte)* wird der Zustand des Spiels aktualisiert. Dies geht ansatzweise wie folgt:

```
if (Zustand == Zustaende.RotIstAmZug) {
    spielbrett.SpieleStein(Spieler.Rot,
                           spalte);
    Zustand = Zustaende.GelbIstAmZug;
}
```

Es wird also ein entsprechender Spielstein gelegt und anschließend ermittelt, wer am Zug ist. Etwas später folgt dann die Auswertung eines möglichen Gewinners:

```
if (spielbrett.Gewinner == Spieler.Rot){
    Zustand = Zustaende.RotHatGewonnen;
}
```

Die Ermittlung eines Gewinners erfolgt also im Spielbrett, während hier nur der Zustand des Spiels verwaltet wird.

Fazit: Die richtigen Vorüberlegungen sind der Schlüssel zu einer erfolgreichen Implementierung. **[ml]**

[1] Stefan Lieser, Wer übt, gewinnt, dotnetpro 3/2010, Seite 118 f., www.dotnetpro.de/A1003dojo

Listing 2

Prüfen auf gleiche Farben.

```
internal static IEnumerable<IVierer> SelbeFarbe(this IEnumerable<IVierer> vierer,
int[,] feld) {
    foreach (var vier in vierer) {
        if ((feld[vier.Eins.X, vier.Eins.Y] != 0) &&
            (feld[vier.Eins.X, vier.Eins.Y] == feld[vier.Zwei.X, vier.Zwei.Y]) &&
            (feld[vier.Eins.X, vier.Eins.Y] == feld[vier.Drei.X, vier.Drei.Y]) &&
            (feld[vier.Eins.X, vier.Eins.Y] == feld[vier.Vier.X, vier.Vier.Y])) {
            yield return vier;  } } }
```

INotifyPropertyChanged-Logik automatisiert testen

Zauberwort

DataBinding ist eine tolle Sache: Objekt an Formular binden und wie von Zauberhand stellen die Controls die Eigenschaftswerte des Objekts dar. DataBinding ist aber auch knifflig. Stefan, kannst du dazu eine Aufgabe stellen?

DataBinding ist beliebt. Lästig daran ist: Man muss die *INotifyPropertyChanged*-Schnittstelle implementieren. Sie fordert, dass bei Änderungen an den Eigenschaften eines Objekts das Ereignis *PropertyChanged* ausgelöst wird. Dabei muss dem Ereignis der Name der geänderten Eigenschaft als Parameter in Form einer Zeichenkette übergeben werden. Die Frage, die uns diesmal beim dotnetpro.dojo interessiert, ist: Wie kann man die Implementierung der *INotifyPropertyChanged*-Schnittstelle automatisiert testen?

Die Funktionsweise des Events für eine einzelne Eigenschaft zu prüfen ist nicht schwer. Man bindet einen Delegate an den *PropertyChanged*-Event und prüft, ob er bei Änderung der Eigenschaft aufgerufen wird. Außerdem ist zu prüfen, ob der übergebene Name der Eigenschaft korrekt ist, siehe Listing 3.

Um zu prüfen, ob der Delegate aufgerufen wurde, erhöhen Sie im Delegate beispielsweise eine Variable, die außerhalb definiert ist. Durch diesen Seiteneffekt können Sie überprüfen, ob der Event beim Ändern der Eigenschaft ausgelöst und dadurch der Delegate aufgerufen wurde. Den Namen der Eigenschaft prüfen Sie innerhalb des Delegates mit einem *Assert*.

Solche Tests für jede Eigenschaft und jede Klasse, die *INotifyPropertyChanged* implementiert, zu schreiben, wäre keine Lösung, weil Sie dabei Code wiederholen würden. Da die Eigenschaften einer Klasse per Reflection ermittelt werden können, ist es nicht schwer, den Testcode so zu verallgemeinern, dass damit alle Eigenschaften einer Klasse getestet werden können. Also lautet in diesem Monat die Aufgabe: Implementieren Sie eine Klasse zum automatisierten Testen der *INotifyPropertyChanged*-Logik. Die zu implementierende Funktionalität ist ein Werkzeug zum Testen von ViewModels. Dieses Werkzeug soll wie folgt bedient werden:

```
NotificationTester.Verify<MyViewModel>();
```

Die Klasse, die auf *INotifyPropertyChanged*-Semantik geprüft werden soll, wird als generischer Typparameter an die Methode übergeben. Die Prüfung soll so erfolgen, dass per Reflection alle Eigenschaften der Klasse gesucht werden, die über einen Setter und Getter verfügen. Für diese Eigenschaften soll geprüft werden, ob sie bei einer Zuweisung an die Eigenschaft den *PropertyChanged*-Event auslösen und dabei den Namen der Eigenschaft korrekt übergeben. Wird der Event nicht korrekt ausgelöst, muss eine Ausnahme ausgelöst werden. Diese führt bei der Ausführung des Tests durch das Unit-Test-Framework zum Scheitern des Tests.

Damit man weiß, für welche Eigenschaft die Logik nicht korrekt implementiert ist, sollte die Ausnahme mit den notwendigen Informationen ausgestattet werden, also dem Namen der Klasse und der Eigenschaft, für die der Test fehlschlug.

In einer weiteren Ausbaustufe könnte das Werkzeug dann auch auf Klassen angewandt werden, die ebenfalls per Reflection ermittelt wurden. Fasst man beispielsweise sämtliche ViewModels in einem bestimmten Namespace zusammen, kann eine Assembly nach ViewModels durchsucht werden. Damit die so gefundenen Klassen überprüft werden können, muss es möglich sein, das Testwerkzeug auch mit einem Typ als Parameter aufzurufen:

```
NotificationTester.Verify
   (typeof(MyViewModel));
```

Im nächsten Heft finden Sie eine Lösung des Problems. Aber versuchen Sie sich zunächst selbst an der Aufgabe. **[ml]**

Wer übt, gewinnt

In jeder dotnetpro finden Sie eine Übungsaufgabe von Stefan Lieser, die in maximal drei Stunden zu lösen sein sollte. Wer die Zeit investiert, gewinnt in jedem Fall – wenn auch keine materiellen Dinge, so doch Erfahrung und Wissen.

Es gilt:

- Falsche Lösungen gibt es nicht. Es gibt möglicherweise elegantere, kürzere oder schnellere Lösungen, aber keine falschen.
- Wichtig ist, dass Sie reflektieren, was Sie gemacht haben. Das können Sie, indem Sie Ihre Lösung mit der vergleichen, die Sie eine Ausgabe später in dotnetpro finden.

Übung macht den Meister. Also – los geht's. Aber Sie wollten doch nicht etwa sofort Visual Studio starten …

Listing 3

Property changed?

```
[Test]
public void Name_Property_loest_PropertyChanged_Event_korrekt_aus() {
    var kunde = new Kunde();
    var count = 0;
    kunde.PropertyChanged += (o, e) => {
        count++;
        Assert.That(e.PropertyName, Is.EqualTo("Name"));
    };
    kunde.Name = "Stefan"; Assert.That(count,Is.EqualTo(1));
}
```