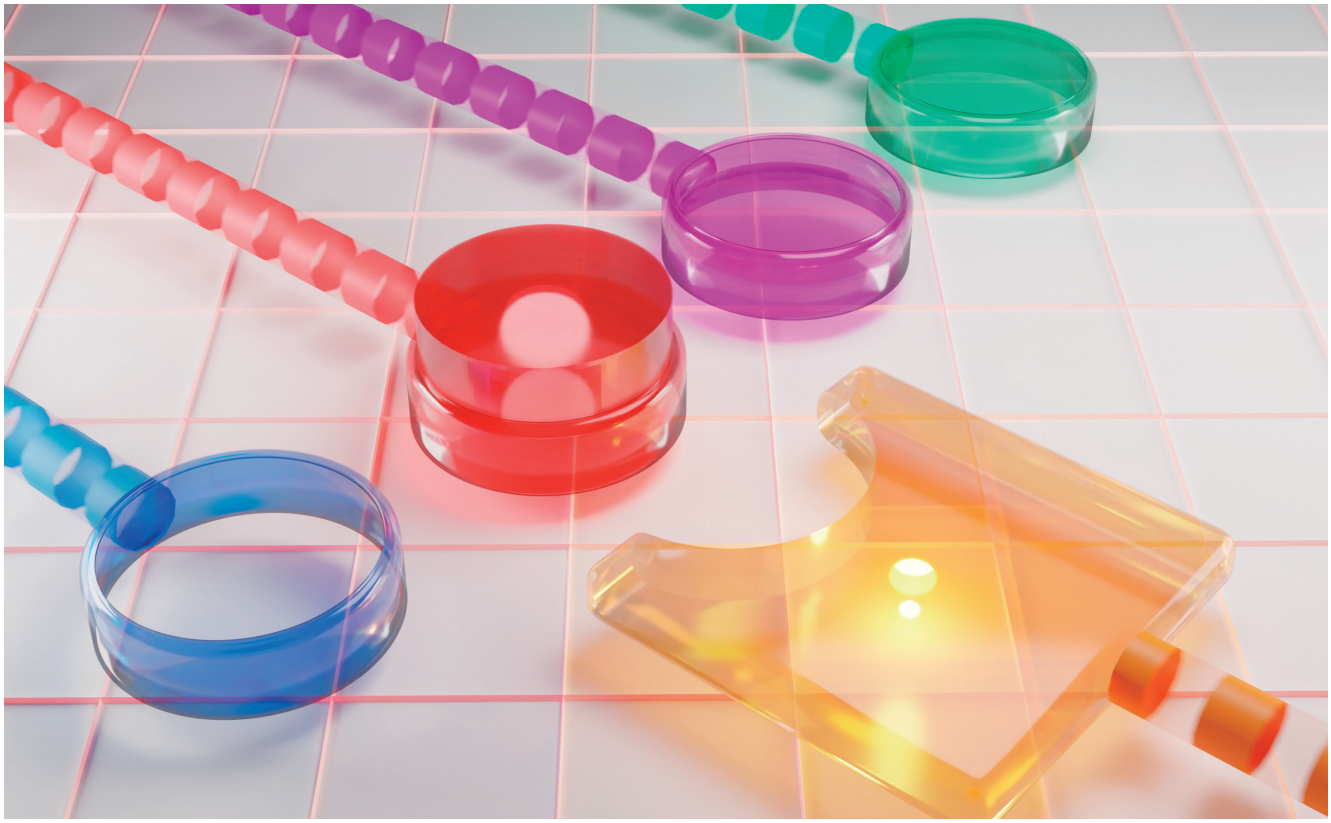
PRINZIPIEN VON DEPENDENCY-INJECTION-CONTAINERN

Entmystifiziert

Für Dependency Injection gibt es viele Bibliotheken. Aber wie funktionieren die eigentlich?



In Seminaren zu Technologien, die Dependency Injection (DI) einsetzen, ist immer wieder festzustellen, dass die Teilnehmer zwar einerseits schnell den Umgang damit lernen, das Grundverständnis für die Zusammenhänge aber andererseits fehlt. Eine Aussage wie „Das ist schon irgendwie magisch, wie der Konstruktor meiner Klasse von Zauberhand ausgeführt wird und die Parameter gesetzt werden" ist durchaus des Öfteren zu hören.

Aber ist das wirklich so magisch? Oder lässt sich Dependency Injection vielleicht sogar mit wenigen Codezeilen umsetzen? Das wäre doch einmal auszuprobieren.

Dependency Injection – was ist das überhaupt?

Zur Entkopplung von Deklaration und Implementierung einer Klasse setzt man häufig auf Schnittstellen (Interfaces). Sie beschreiben in Form eines Vertrags, was die implementierende Klasse auf jeden Fall umsetzen muss. Eine Referenz vom Typ des Interfaces verschafft dann Zugriff auf die benötigten Methoden und Eigenschaften, ohne deren Implementierung kennen zu müssen. Die tatsächlichen Klasseninstanzen sind jederzeit austauschbar, zum Beispiel für Unit-Tests, Technologievarianten für Datenzugriffe und so weiter. Allerdings müssen die Objekte ja irgendwo instanziert werden, und dafür sind dann doch die konkreten Klassen nötig; zudem muss man genau wissen, wie die betreffenden Konstruktoren mit Parametern zu versorgen sind.

Hier kommt Dependency Injection ins Spiel. Die Idee ist, in einem Container in einer ersten Phase Datentypen anzumelden, von denen sich später in der zweiten Phase Instanzen abrufen lassen. Für die Registrierung sind im Wesentlichen zwei Dinge notwendig:

- ein eindeutiger Schlüssel
- und eine Factory-Methode zum Anlegen oder Abrufen einer Instanz.

Als Schlüssel dient in den meisten Fällen ein Typobjekt (Instanz der Klasse *Type*). Der zu instanzierende Typ muss dann entweder mit diesem Typ identisch sein (das wäre der einfachste Fall), oder er muss den Schlüsseltyp implementieren (durch Interface-Implementierung oder Ableitung von der Klasse, die den Schlüssel repräsentiert). Über diesen Schlüssel lässt sich dann später die zugeordnete Factory-Methode abrufen. Das Ausführen dieser Methode liefert als Ergebnis die Referenz des gewünschten Objekts.

Wie die Factory-Methode aussehen muss, hängt wiederum davon ab, wie die Instanzierung erfolgen soll. Hier lassen sich wieder grundsätzlich zwei Varianten unterscheiden:

- als Singleton
- transient

Mit dieser Unterscheidung wird die Lebensdauer festgelegt. Ein Singleton soll nur einmal instanziert werden und bleibt dann für die Laufzeit der Anwendung im Speicher. Die Instanzierung kann bereits bei der Registrierung erfolgen oder beim ersten Abruf.

Transiente Objekte hingegen werden bei jedem Abruf neu erzeugt und können nach Gebrauch wieder freigegeben werden. Die Entscheidung, ob Datentypen als Singletons oder transiente Objekte registriert werden sollen, hängt vom jeweiligen Einsatzfall ab.

Spezifische DI-Implementierungen stellen noch weitere Varianten bereit, die aber letztlich auf diesen beiden Typen basieren. So kennt .NET noch den sogenannten Scoped-Typ, der zum Beispiel in ASP.NET verwendet wird, um DI-Abrufe, die zu derselben Abarbeitung einer Anfrage gehören, wie ein Singleton zu betrachten, diese nach Ende der Bearbeitung aber wieder verwirft.

Grundgerüst

Die obige Beschreibung lässt schon erahnen, was für den Aufbau eines Grundgerüsts nötig ist: eine Schlüssel-Wert-Liste, bei der der Schlüssel (*Key*) das *Type*-Objekt ist und *Value* die zugehörige Factory-Methode. Bild 1 stellt die Zusammenhänge grafisch dar. Zu jedem *Type*-Objekt (*Key*) gibt es eine Methode (*Value*), über die der Zugriff auf die gewünschten Objekte erfolgt. Je nach Factory-Methode erzeugt jeder Aufruf ein neues Objekt (transient) oder verweist stets auf dasselbe, einmalig instanzierte (Singleton).

Listing 1 zeigt eine erste Implementierung hierzu in Form der Container-Klasse *ServiceLocator*. Sie verwendet intern ein Dictionary zum Speichern der Registrierungen. Der Schlüssel ist wie beschrieben ein *Type*-Objekt, der Wert eine parameterlose Funktion, die eine *object*-Referenz zurückgibt. *ServiceLocator* ist als Singleton implementiert (privater Konstruktor, statische Eigenschaft *Current* für den Zugriff auf die Instanz).

Die einfachste Variante zum Registrieren eines Singletons nimmt die Referenz eines bestehenden Objekts entgegen, ermittelt das Typobjekt, erstellt eine einfache Factory-Methode, welche die Referenz zurückgibt, und speichert dieses Paar im Dictionary. Für den Typ *S* (siehe Beispieltypen in Listing 2) könnte die Registrierung dann mit folgendem Aufruf erfolgen:

```
ServiceLocator.Current.AddSingleton(new S());
```

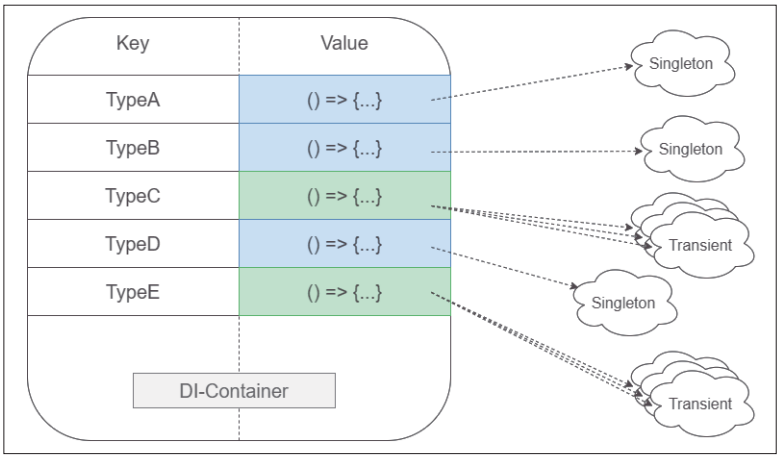
Eine zweite Variante der Methode erlaubt es, den Typ für den Schlüssel abweichend vom Instanztyp festzulegen. In der Methode müsste dann zur Laufzeit geprüft werden, ob das angegebene Objekt auch tatsächlich den Schlüsseltyp implementiert. Diese und weitere Prüfungen sind aus Gründen der Übersichtlichkeit im Beispielcode nicht enthalten.

Zum Abrufen der registrierten Typen stehen die beiden Überladungen von *GetInstance()* zur Verfügung. Die erste nimmt den Schlüsseltyp als generischen Typparameter entgegen und gibt die Instanz auch typisiert zurück; die zweite arbeitet nichtgenerisch mit *object*-Referenzen. Das Einzige, was diese Methoden tun müssen, ist, über den Schlüsseltyp die Factory-Methode aus dem Dictionary zu holen und sie auszuführen. Im Beispiel der obigen Registrierung von *S* könnte ein Aufruf dann so aussehen:

```
var s = ServiceLocator.Current.GetInstance<S>();
Console.WriteLine(s.Name);
```

Das führt dann zur Ausgabe *ein S*.

Etwas komplizierter wird es, wenn das Objekt bei der Registrierung noch nicht existiert und erst beim ersten Abruf erzeugt werden soll. Darauf wird weiter unten noch eingegangen; zunächst soll aber die Registrierung von transienten Typen betrachtet werden.



Ein DI-Container ist eine Schlüssel-Wert-Liste mit Factory-Methoden zum Erstellen und Abrufen der Objekte (Bild 1)

Vergängliches

Die Registrierung von transienten Typen erfolgt auf ähnliche Weise. In Listing 3 sind zwei generische Varianten zu sehen, die Schlüssel- und Instanztyp entweder unterscheiden oder den Instanztyp gleichzeitig auch als Schlüssel verwenden. Da die Instanzierung etwas aufwendiger, aber immer wieder gleich ist, wurde sie in die Methode *CreateInstance()* ausgelagert. Die jeweiligen Factory-Methoden rufen sie auf und reichen das Ergebnis durch.

CreateInstance() nutzt Reflection, um einen geeigneten Konstruktor zu finden und dessen Parameterwünsche zu erfüllen. Der Aufruf von *t.GetConstructors().Single()* erzwingt dabei, dass es genau einen öffentlichen Konstruktor gibt. Ist keiner vorhanden oder gibt es mehr als einen, führt der Aufruf zu einer Ausnahme, die in der Demo nicht weiter be- ▶

handelt wird. *ctor.GetParameters()* liest die Parameterangaben dieses Konstruktors.

Als Parametertyp kommen nur Typen infrage, die bereits in der Registrierungsphase dem Container bekannt gemacht worden sind. Deren Werte lassen sich daher einfach durch Aufruf der in Listing 1 gezeigten Methode *GetInstance()* abrufen. Wurden alle Parameterwerte ermittelt, kann die Instanz via *ctor.Invoke(paramValues)* erzeugt werden.

Nehmen wir den Fall einer Beispielklasse *A*, die das Interface *IA* implementiert. Die Registrierung könnte so aussehen:

```
ServiceLocator.Current.AddTransient<IA, A>();
```

Ein Aufruf von

```
var a = ServiceLocator.Current.GetInstance<IA>();
a.TuWas();
```

liefert als Ausgabe:

```
TuWas A
```

Der Konstruktor der Klasse *A* hat keine Parameter, und so beschränkt sich die Aufgabe von *CreateInstance()* auf dessen Ausführung.

Im Fall der Klasse *B* ist nun allerdings ein wenig mehr zu tun:

```
ServiceLocator.Current.AddTransient<IB, B>();
...

var b = ServiceLocator.Current.GetInstance<IB>();
b.MachWas();
```

Der Konstruktor der Klasse *B* erwartet einen Parameter vom Typ *IA* und einen vom Typ *S*. Diese beide müssen entsprechend aufgelöst werden, sodass die Prozedur *GetInstance()* hier zweimal zum Einsatz kommt. Als Ausgabe ergibt sich in diesem Fall:

Listing 1: Ein erster Ansatz

```
public class ServiceLocator
{
  // Key: registrierter Typ, Value: Factory-Function
  // zum Anlegen eines Objekts
  private Dictionary<Type,
    Func<object>> container = new();

  // Singleton-Konstruktion der
  // ServiceLocator-Klasse
  public static readonly ServiceLocator Current =
    new ServiceLocator();
  private ServiceLocator() { }

  // Methoden zum Aufbauen des DI-Containers

  // Hinzufügen eines vorhandenen Objekts als
  // Singleton
  // obj: Das bereits instanzierte Objekt. Der Typ
  // wird als Schlüssel verwendet
  public void AddSingleton(object obj)
  {
    container.Add(obj.GetType(), () => obj);
  }

  // Hinzufügen eines vorhandenen Objekts als
  // Singleton
  // TKey: Typ, für den das Objekt später abgerufen
  // werden soll.
  // obj: Das zu registrierende Objekt. Muss TKey
  // implementieren</param>
  public void AddSingleton<TKey>(object obj)
  {
    // hier sollte noch geprüft werden, ob obj
    // wirklich TKey implementiert
    container.Add(typeof(TKey), () => obj);
  }

  ...

  // Methoden zum Abrufen der Instanzen

  // Abrufen eines Objekts
  // T: Schlüsseltyp
  // Rückgabe: Das (vorhandene oder generierte)
  // Objekt
  public T GetInstance<T>()
  {
    // Abruf der Factory-Methode aus dem
    // Dictionary und ausführen derselben
    return (T)container[typeof(T)]();
  }

  // Abrufen eines Objekts
  // t: Schlüsseltyp
  // Rückgabe: Das (vorhandene oder generierte)
  // Objekt
  public object GetInstance(Type t)
  {
    // Abruf der Factory-Methode aus dem
    // Dictionary und ausführen derselben
    return container[t]();
  }

}
```

```
B Machwas mit ein S
TuWas A
```

Doch was passiert, wenn der Typ *IC* angefordert wird? Die Implementierung von *C* erwartet einen Konstruktorparameter vom Typ *IB*, der aber erst konstruiert werden muss. Die bisherige Implementierung löst das bereits auf. In diesem Fall wird *GetInstance()* so oft rekursiv aufgerufen, bis alle Parameterwerte verfügbar sind. Die Aufruffolge von

```
ServiceLocator.Current.AddTransient<IC, C>();
...
var c = ServiceLocator.Current.GetInstance<IC>();
c.Ausgabe();
```

liefert demzufolge als Ausgabe: ▶

Listing 2: Beispielklassen

```
public class S
{
  public string Name { get; set; } = "ein S";
}

public interface IA
{
  void TuWas();
}

public class A : IA
{
  public void TuWas()
  {
    Console.WriteLine("TuWas A");
  }
}

public interface IB
{
  void MachWas();
}

public class B : IB
{
  private readonly IA a;
  private readonly S s;

  public B(IA a, S s)
  {
    this.a = a;
    this.s = s;
  }
  public void MachWas()
  {
    Console.WriteLine("B Machwas mit " + s.Name);
    a.TuWas();
  }
}

public interface IC { void Ausgabe(); }
public class C : IC
{
  private readonly IB b;

  public C(IB b)
  {
    this.b = b;
  }
  public void Ausgabe()
  {
    Console.WriteLine("Ausgabe C");
    b.MachWas();
  }
}

public class D
{
  private readonly IB b;

  public D(IB b)
  {
    this.b = b;
  }
  public void Ausgeben()
  {
    Console.WriteLine("ctor D");
    b.MachWas();
  }
}

public interface IE { void Print(); }
public class E : IE
{
  private readonly D d;

  public E(D d)
  {
    this.d = d;
  }
  public void Print()
  {
    Console.WriteLine("Print E");
    d.Ausgeben();
  }
}
```

```
Ausgabe C
B Machwas mit ein S
TuWas A
```

An dieser Stelle sei erwähnt, dass nach dem Aufbau des Containers eine Plausibilitätsprüfung erfolgen muss, bevor Instanzen abgerufen werden können. Zum Beispiel könnte es sein, dass bei den registrierten Typen zirkuläre Abhängigkeiten einbaut sind. Wenn also beispielsweise der Konstruktor von *A* einen Parameter vom Typ *IB* erwartet, der von *B* aber wie im Beispiel einen Parameter vom Typ *IA* hat, dann würde die Implementierung von *GetInstance()* zu einer hässlichen *StackOverflowException* führen.

Solche zirkulären Referenzen können auch etwas versteckter über mehrere Hierarchien hinweg erfolgen, sodass eine Vorabprüfung durchaus sinnvoll ist. Auch könnten Typen registriert werden, die sich aus anderen Gründen nicht instanzieren lassen. DI-Frameworks haben hierzu meist eine Build-Methode, die den Abschluss der Registrierungsphase signalisiert und die notwendigen Prüfungen vornimmt. Hier soll es aber nur darum gehen, das Prinzip des DI-Containers zu erklären, nicht um die Fleißarbeit der Prüfungen (die noch mehr Reflection-Akrobatik erfordert).

Und die Singletons mit späterer Instanzierung?

Die spätere Instanzierung ist nun auch keine Zauberei mehr. Im Grunde ist nur ein weiteres Dictionary nötig, um die einmal instanzierten Objekte vorhalten zu können (siehe Listing 4). In der generischen Methode *AddSingleton<TKey, TValue>()* wird die Factory-Methode so implementiert, dass sie prüft, ob die gewünschte Instanz bereits im Dictionary *singletons* referenziert ist. Falls ja, wird diese Referenz zurückgegeben, falls nein, wird wieder mittels *CreateInstance()* eine neue Instanz erzeugt, im Dictionary verlinkt und deren Referenz zurückgegeben.

AddSingleton<TValue>() ist wiederum eine vereinfachte Variante, bei welcher der generische Typ gleichzeitig Schlüssel- und Instanztyp ist. Sie lässt sich einfach in die andere Variante überführen.

Das folgende Beispiel registriert *D* und ruft es später ab:

```
ServiceLocator.Current.AddSingleton<D>();
...
var d1 = ServiceLocator.Current.GetInstance<D>();
d1.Ausgeben();
var d2 = ServiceLocator.Current.GetInstance<D>();
Console.WriteLine(d1==d2);
```

Das führt folglich zu dieser Ausgabe:

```
ctor D
B Machwas mit ein S
TuWas A
True
```

Listing 3: Registrieren von transienten Typen

```
public class ServiceLocator
{
  ...
  // Registrieren eines Typs als Transient
  // TKey: Schlüsseltyp für den späteren Abruf
  // TValue: Zu instanzierender Typ (muss TKey
  // implementieren)
  public void AddTransient<TKey, TValue>()
      where TValue : TKey
  {
    container.Add(typeof(TKey), () =>
      CreateInstance<TValue>());
  }

  // Registrieren eines Typs als Transient
  // TValue: Der zu registrierende Typ. Dieser wird
  // auch als Key verwendet
  public void AddTransient<TValue>()
  {
    AddTransient<TValue, TValue>();
  }

  ...

  // Erzeugen einer Instanz vom Typ TValue
  private TValue CreateInstance<TValue>()
  {
    // zu instanzierender Typ
    var t = typeof(TValue);

    // es muss genau einen Konstruktor geben
    var ctor = t.GetConstructors().Single();

    // Parameter dieses Konstruktors ermitteln und
    // Werte hierfür holen
    var parameters = ctor.GetParameters();
    var paramValues =
      new object[parameters.Length];
    for (int i = 0; i < parameters.Length; i++)
    {
      paramValues[i] = GetInstance(
        parameters[i].ParameterType);
    }

    // Instanz erzeugen und zurückgeben
    return (TValue)ctor.Invoke(paramValues);
  }
}
```

Listing 4: Registrieren von Singleton-Typen

```
public class ServiceLocator
{
  // Liste der bereits instanzierten Singletons
  private Dictionary<Type, object>
    singletons = new();

  // Registrieren eines Typs als Singleton
  // TValue: Der zu registrierende Typ. Dieser wird
  // auch als Key verwendet
  public void AddSingleton<TValue>()
  {
    AddSingleton<TValue, TValue>();
  }

  // Registrieren eines Typs als Singleton
  // TKey: Schlüsseltyp für den späteren Abruf
  // TValue: Zu instanzierender Typ (muss TKey
  // implementieren)
  public void AddSingleton<TKey, TValue>()
      where TValue : TKey
  {
    var tKey = typeof(TKey);
    container.Add(tKey, () =>
    {
      // Objekt schon vorhanden?
      if (singletons.ContainsKey(tKey))
      return singletons[tKey];

      // Neue Instanz anlegen und speichern
      var obj = CreateInstance<TValue>();
      singletons.Add(tKey, obj);
      return obj;
    });
  }
  ...
}
```

Der erste Abruf via *GetInstance()* erzeugt hier die Instanz, alle nachfolgenden Aufrufe liefern dann immer dieselbe Referenz zurück. Beim Vergleich von *d1* und *d2* ergibt sich daher der Wert *True*.

Der letzte Testlauf zeigt die Registrierung von *E* als Singleton mit *IE* als Schlüssel:

```
ServiceLocator.Current.AddSingleton<IE, E>();
...
var e = ServiceLocator.Current.GetInstance<IE>();
e.Print();
```

Die Ausgabe lautet:

```
Print E
ctor D
B Machwas mit ein S
TuWas A
```

Was folgt, ist Fleißarbeit

Das Grundgerüst des DI-Containers steht somit. Es lassen sich Datentypen als Singleton oder transiente Objekte registrieren und über die *GetInstance()*-Methoden später abrufen. Damit ist schon die ganze Magie der Dependency Injection entzaubert und umgesetzt.

Wie bereits mehrfach erwähnt, fehlen dem Beispielcode noch viele wichtige Prüfungen, die ein fertiger DI-Container natürlich leisten muss. Multithreading kann ebenso ein wichtiges Thema sein; die Beispielimplementierung ist bestimmt nicht Thread-sicher.

Funktionsumfang und Komfort lassen sich beliebig erweitern. Manche Container unterstützen Mehrfachregistrierungen zu einem Typ. Dann müssen in der Schlüssel-Wert-Liste zu jedem Eintrag mehrere Factory-Methoden gespeichert werden. Überladene Zugriffsmethoden (*GetInstance()*) liefern dann nicht ein einzelnes Objekt, sondern eine Liste von Objekten. Auch kann man sich viele weitere lustige Überladungsvarianten für die Registrierungsmethoden ausdenken. Oder eine Lösung vorsehen, die eine externe Assembly lädt und automatisch bestimmte Datentypen dieser Assembly im Container registriert; oder, oder, oder. Aber dann hat man das vielleicht 156.017. DI-Framework programmiert.

Fazit

Die wenigen Codezeilen der Klasse *ServiceLocator* haben gezeigt, dass es vergleichsweise einfach ist, die Idee der Dependency Injection umzusetzen. Ein paar Schlüssel-Wert-Listen, ein paar Lambda-Ausdrücke, etwas Generics, etwas Reflection – das war's auch schon. Zwar fällt die Implementierung realer DI-Frameworks sicher etwas umfangreicher aus, aber letztlich basieren sie alle auf den gezeigten Prinzipien.

Falls Sie selbst etwas experimentieren möchten: Der Code zum Beispielprojekt ist bei GitHub zu finden [1]. ■

[1] Quellcode auf GitHub, www.dotnetpro.de/SL2111DI1

Dr. Joachim Fuchs
ist begeisterter Anhänger von Microsofts .NET-Philosophie. Er arbeitet als Softwarearchitekt, Berater und Dozent im Expertennetzwerk www.it-visions.de. Seine Schwerpunkte liegen derzeit bei XAML- und Web-UI-Technologien.
dnp@fuechse-online.de

dnpCode A2111DI