Arbeiten mit Java-Schnittstellen und -Klassen

Teil 1 von 2

Dirk Riehle, Ubilab, UBS

Bahnhofstrasse 45, CH-8021 Zürich

Schnittstellen sind in der Softwaretechnik seit über 25 Jahren bekannt. Objektorientierte Sprachen wie C++ und Smalltalk unterscheiden allerdings nicht zwischen Schnittstellen und Klassen, die diese Schnittstellen implementieren. Java hingegen bietet diese Unterscheidung an, die allerdings für viele Java-Entwickler anfangs ungewohnt ist. Dieser Artikel zeigt daher auf, wie man Schnittstellen und Klassen in der Modellierung und Implementierung von Java-basierten Systemen unterscheidet und konstruktiv einsetzen kann. Der vorliegende erste Teil befaßt sich zunächst mit den Grundlagen, die an dem Beispiel der Implementierung eines einfachen Namensdienstes illustriert werden.

1 Einleitung

Java bietet neben dem Klassenkonzept auch die Verwendung von Schnittstellen (interfaces) zur Implementierung von Softwaresystemen an. Auf der einen Seite ist die Unterscheidung von Schnittstellen und Implementierungen ein in der Softwaretechnik lang bekanntes und unbestritten wichtiges Konzept. Auf der anderen Seite zeigen prominente Beispiele, daß der sinnvolle Einsatz von Schnittstellen in Java nicht immer selbstverständlich ist.

In diesem und einem weiteren Artikel beschreibe ich die Bedeutung von Schnittstellen und ihre Unterscheidung von Klassen zur Java-basierten Implementierung von Softwaresystemen. Ich zeige auf, wie die drei Konzepte Schnittstelle, abstrakte Klasse und konkrete Klasse zusammenarbeiten können, um ihnen bei der Implementierung von Softwaresystemen zu helfen und ihre Anwendungen flexibler und leichter änderbar zu machen.

Ich bediene mich dabei eines einfachen aber häufig zu findenden Beispiels: den Entwurf eines Namensdienstes (naming service), wie er in der CORBA Object Services Spezifikation (COSS) definiert wird.

Insgesamt gibt es zwei Teile. Dieser erste Artikel führt die grundlegenden Konzepte zur Unterscheidung von Schmittstellen und Klassen ein. Der zweite Artikel beschreibt fortgeschrittene Techniken zur Implementierung von Schnittstellen. Dies betrifft insbesondere die Organisation von Klassenhierarchien, um die Codewiederverwendung zu maximieren.

2 Klassen und Schnittstellen in Entwurf und Implementierung

Objektorientierung ist bekanntermaßen ein Denk-Paradigma, das gleichermaßen für die Analyse, den Entwurf und die Implementierung von Softwaresystemen verwendet werden kann. Hierzu bietet es eine Menge von Konzepten an, denen eine unterschiedliche Bedeutung zukommt, je nachdem ob man sie in der Analyse, im Entwurf oder in der Implementierung verwendet.

Im angelsächsischen Sprachraum wird der Begriff Objekt zumeist als Kapsel um gemeinsame Daten und Operationen definiert. Im skandinavischen Sprachraum wird der Begriff Objekt als ein in Anwendungsbereichen anzutreffendes Phänomen definiert. Ähnlich sieht es mit dem Konzept der Klasse aus: Für die einen ist es eine Schablone zum Erzeugen von Objekten, für die anderen ist es die Abstraktion von gleichartigen Objekten, die wir in Anwendungsbereichen vorfinden.

Offenkundig handelt es sich hierbei um komplementäre Definitionen, eine technische aus der Entwurfs- und Implementierungswelt, und eine anwendungsfachliche aus der Analyse- und Entwurfswelt. Gute Methodologien machen dies auch klar. Diese Konzepte, ihre Definitionen und ihre entsprechende Anwendung existieren zumeist friedlich nebeneinander und harmonieren auch gut.

Die skandinavische Schule rund um die objektorientierten Programmiersprachen SIMULA 67 und Beta hat immer wieder versucht, alle drei Bereiche, Analyse, Entwurf und Implementierung mit denselben Konzepten beschreibbar

zu machen: eine Notation für alle Belange. Da SIMULA 67 als erste objektorientierte Programmiersprache der Welt gelten kann, haben diese Bemühungen ihren Einfluß hinterlassen, und so kennen Programmiersprachen wie C++ und Smalltalk als hauptsächliches objektorientiertes Programmierkonzept nur die Klasse.

Java geht nun einen Schritt weiter und erweitert die Menge der Programmierkonzepte um den wohlbekannten Begriff der Schnittstelle. Zwar sprach man etwa in C++ oder in Smalltalk bisher auch schon davon, daß eine Klasse eine Schnittstelle besitzt, diese war aber nie von der Klasse selbst und ihrer Implementierung getrennt. In Java kann man nun Schnittstellen definieren, ohne gleich eine Implementierung mit angeben zu müssen. In C++ hatte man dazu immer auf Hilfsmittel wie abstrakte Klassen mit ausschließlich als “pure virtual” deklarierten Operationen zurückgreifen müssen.

Schnittstellen sind kein neues Konzept: Sie sind in der Softwaretechnik seit über 25 Jahren wohlbekannt. Ihre Aufgabe ist es, die Zugriffsmöglichkeiten auf Objekte, die diese Schnittstelle erfüllen, zu definieren und es zu ermöglichen, daß die Implementierung dieser Objekte variieren kann, ohne daß von der Schnittstelle abhängige Klienten sich ändern müßten. Java-Schnittstellen gruppieren eine Menge zusammengehöriger Operationen, ohne etwas über ihre Implementierung zu sagen. Schnittstellen können mittels Mehrfacherben gebildet werden, während Klassen lediglich auf einfacher Vererbung basieren.

In Java neu ist die Möglichkeit, Schnittstellen durch Erben von mehreren (nicht nur einer) Schnittstellen zu bilden. Dies ermöglicht es, ein Objekt aus mehreren Sichten, daß heißt über unterschiedliche Schnittstellen, zu betrachten. Dies kann zum Beispiel dazu verwendet werden, um eine Schnittstelle durch das Erben von Protokollschnittstellen zu definieren [1], welche jeweils spezifische technische oder fachliche Einzelaspeke beschreiben, die aber nicht für sich alleinstehend sinnvoll wären.

In Java implementiert man Schnittstellen durch Klassen. Man zeigt dies durch die “implements”-Beziehung an. Objekte einer Klasse, welche eine bestimmte Schnittstelle implementiert, können in allen Kontexten verwendet werden, wo ein Klient diese Objekte unter der implementierten Schnittstelle erwartet.

Die Einführung von Schnittstellen auf Programmierebene ist ein wichtiger Fortschritt gegenüber industriellen Programmiersprachen, in denen diese Unterscheidung nicht explizit getroffen werden kann. Er ist auch gleichzeitig verwirrend, denn in der Analyse arbeitet man weiterhin mit Klassen: Im Bankbereich definiert man zum Beispiel die Klassen Person und Konto, im Entwurf oder in der Programmierung mit Java aber drückt man diese Analyseklassen durch eine Schnittstelle aus! Somit bildet man Analyseklassen auf Java-Schnittstellen ab, und Java-Klassen werden lediglich als Implementierungen dieser Schnittstellen verwendet.

Im folgenden werde ich die Konsequenzen dieser Trennung zwischen Schnittstelle und Klasse auf Programmierebene genauer untersuchen und mit Beispielen unterlegen. Insbesondere beschreibe ich verschiedene Entwurfs- und Programmiermuster, welche zeigen, wie man nicht nur mit Schnittstellen konzeptuell klar einen Entwurf abbildet, sondern auch mithilfe von abstrakten Klassen zu guter Codewiederverwendung gelangt.

3 Schnittstellen und Implementierungen

Ich verwende als durchgängiges Beispiel den Entwurf und die Implementierung eines Namensdienstes (naming service), wie er in der OMG CORBA Object Services Spezifikation (OMG-COSS) definiert ist [2]. Ein Namensdienst dient dazu, einem Objekt einen Namen zu geben, um es später unter diesem Namen wieder heraussuchen zu können. Objekte gegen Namen heißt die Devise. In diesem Abschnitt beschreibe ich die Implementierung von Namen mit Schnittstellen und konkreten Klassen.

Beispiele für Namen sind die URL “http://www.ubs.com/ubilab” oder der Dateiname “net/projects/Geo/Vroom”. Auf den ersten Blick sind Namen eine einfache Sache, müßte es doch ausreichen, sie durch einen String zu repräsentieren. Tatsächlich aber funktioniert dies nur beschwerlich mit Namen, die eine komplexe innere Struktur besitzen. Die Interpretation solcher Namen sollte besser durch eine eigene Schnittstelle beschrieben werden.

Die folgende Java-Schnittstelle beschreibt eine Namensschnittstelle:

```
public interface Name
{
  public Name context();
  public Name without(Name name);
  public Name extended(String nc);
  public Name context(int end);
  public Name subname(int start, int end);

  public Vector components();
  public int noOfComponents();
  public String component(int i);
  public String lastComponent();
```

```
  public boolean contains(Name name);
}
```

Ein Name besteht aus einer geordneten Liste von Namenskomponenten (laut COSS-Definition). Das heißt, der Name “net/projects/Geo/Vroom” hat die vier Komponenten net, projects, Geo und Vroom. Zur textuellen Darstellung eines Namens als einen einzelnen String verwendet man meist ein Komponententrennzeichen. Im Falle von Dateinamen ist dies unter Unix zum Beispiel der Schrägstrich “/”. Der Einfachheit halber setze ich in diesem Artikel Namenskomponenten mit Strings gleich.

Somit kann man ein Namensobjekt nach einem Vector von Namenskomponenten befragen (public Vector components()). Weiterhin kann man ein Namensobjekt nach seinem Kontextnamen befragen, was im Fall von “net/projects/Geo/Vroom” als Ergebnis den Namen “net/projects/Geo” liefert. In der Umkehrung kann man nach der abschließenden Namenskomponente fragen, was den String “Vroom” liefert. Und so weiter.

Wie implementiert man nun die Schnittstelle Name? Man muß sie mithilfe einer Klasse implementieren. Tatsächlich sind mehrere unterschiedliche Implementierungen denkbar, was uns die Mächtigkeit der Unterscheidung von Klassen und Schnittstellen demonstrieren hilft.

Zum einen kann man Name als einen Vector von Namenskomponenten, also Strings implementieren. Das liefert uns dann die folgende Klasse VectorNameImpl:

```
public class VectorNameImpl
  implements Name
{

public VectorNameImpl()
{
  components = new Vector();
}

public VectorNameImpl(Vector cps)
{
  components = cps;
}

public Name context()
{
  Vector ncps = components.clone();
  ncps.removeElementAt(ncps.size()-1);
  return new VectorNameImpl(ncps);
}

public String lastComponent()
{
  return components.lastElement();
}

... und so weiter

// Implementierungszustand
protected Vector components;

}
```

Die Verwendung eines Vectors von Strings macht es einfach, Teile des Namens als neuen Namen zurückzugeben, oder auch nur die letzte Namenskomponente auf Anfrage zu liefern. Andererseits erscheint die Implementierung eines vermeintlich einfachen Konzepts wie Name mithilfe eines Vectors recht aufwendig. Dies führt uns zu einer alternativen Implementierung.

Genausogut wie mit einem Vector, kann man ein Namensobjekt auch mithilfe eines einzigen Strings implementieren. Der String enthält dann alle Namenskomponenten, fein säuberlich aneinandergereiht, und mit einem Trennzeichen voneinander getrennt. In den Namenskomponenten selbst muß dann natürlich das Trennzeichen maskiert werden. Dies führt uns zur Klasse StringNameImpl, die wie VectorNameImpl auch die Schnittstelle Name implementiert, und von der Objekte erzeugt werden können:

```
public StringNameImpl
  implements Name
{

public StringNameImpl()
{
  nameString = new String();
}

public StringNameImpl(String nstr)
{
  nameString = maskSeparator(nstr);
}

public Name context()
{
```

```
   char sepChar = separatorChar();
   int index = nameString.lastIndexOf(sepChar);
   String contextString = nameString.substring(0, index-1);
   return new StringNameImpl(contextString);
}

public String lastComponent()
{
   char sepChar = separatorChar();
   int index = nameString.lastIndexOf(sepChar);
   return nameString.substring(index, nameString.length());
}

protected char separatorChar()
{
   return '/';
}

protected String maskSeparator(String istr)
{
   // maskiere Trennzeichen in istr ...
}

... und so weiter

// Implementierungszustand
protected String nameString;

}
```

Mit den Klassen VectorNameImpl und StringNameImpl verfügen wir nunmehr über zwei Implementierungen derselben Schnittstelle und können je nach Bedarf die eine oder andere auswählen.

Um Namensobjekte unabhängig von den Klassen, die sie implementieren, zur Laufzeit beliebig vertauschbar zu machen, benötigt man eine Richtlinie. Diese Richtlinie ist bekannt und wurde unter anderem von der Gang-of-four, den Autoren des Buchs "Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software" folgendermaßen formuliert:

Programmiere auf eine Schnittstelle hin, nicht auf eine Implementierung ([3], Seite 21).

Dies bedeutet, immer nur den Namen von Schnittstellen, im Beispiel also Name, zu verwenden. Die Namen der Implementierungsklassen VectorNameImpl oder StringNameImpl werden außer bei der Erzeugung der Objekte nie genannt. Deswegen versehen wir in unseren Projekten auch die Klassen mit dem Postfix "Impl" statt etwa, wie man es gelegentlich sieht, die Schnittstellen mit einem Postfix wie zum Beispiel "I" oder "Ifc" für Interface zu versehen.

Abbildung 1 zeigt die Schnittstelle und ihre zwei Implementierungen auf:

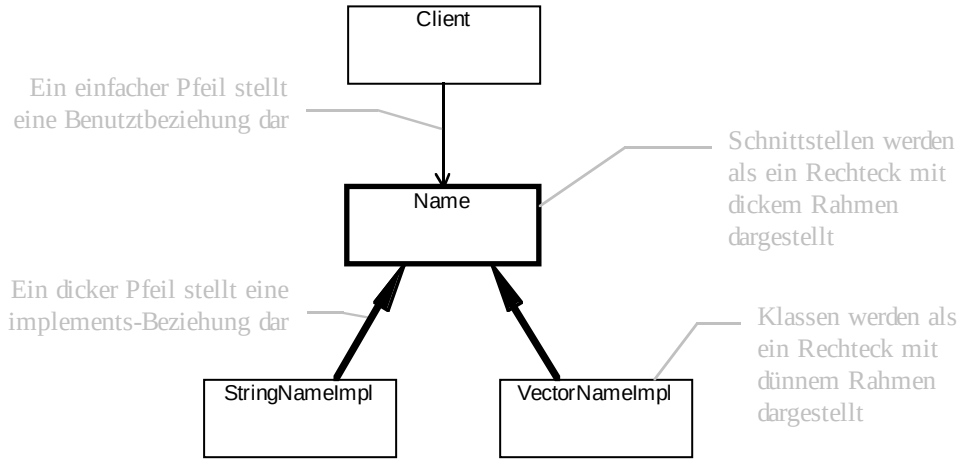

Abbildung 1: Die Schnittstelle von Name mit den zwei Implementierungsklassen StringNameImpl und VectorNameImpl.

In guter objektorientierter Manier kann man natürlich die Erzeugungsoperation, welche die Implementierungsklasse benennt, ebenfalls kapseln. Dies geschieht zum Beispiel mithilfe der Entwurfsmuster abstrakte Fabrik [3] oder Product Trader [4]. Dann tauchen die Implementierungsklassennamen überhaupt nicht mehr im Klientencode auf, wodurch er von ihnen unabhängig wird. Somit kann man die Namensimplementierungen beliebig erweitern und ändern, ohne Klientencode in Mitleidenschaft zu ziehen.

Nach welchen Kriterien wählt man jetzt aber eine bestimmte Implementierung aus? Diese Frage kann nicht allgemein beantwortet werden, sondern hängt immer von den tatsächlich vorhandenen Implementierungen ab. Im konkreten Beispiel wird man VectorNameImpl vielleicht vorziehen, wenn man sehr viel mit den Komponenten eines Namens arbeitet, und somit einen effizienten Zugriff auf die jeweiligen Namenskomponenten haben möchte. StringNameImpl wird man vielleicht dann verwenden, wenn man mit einer hohe Anzahl von Namensobjekten arbeiten möchte, so daß die im Speicherplatzverbrauch effizienteste Implementierung die beste Lösung ist.

Man merkt hier natürlich, daß quasi durch die Hintertür die Unterscheidung der Implementierungen doch noch wichtig wird. Solange diese Unterscheidung aber nur während der Erzeugung der Objekte gebraucht wird und der Klientencode unabhängig davon bleibt, bleiben die Unterschiede der Implementierungen in ihren Auswirkungen zumeist begrenzt.

4 Abstrakte und konkrete Klassen

Der vorige Abschnitt hat die Verwendung von Schnittstellen und ihre einfache Implementierung durch konkrete Klassen geschildert. Um eine gute Codewiederverwendung zu erreichen, müssen wir aber noch darüber nachdenken, wie denn genau Klassen voneinander erben sollten. Dies führt zum Konzept von abstrakten und konkreten Klassen, welches in diesem Abschnitt diskutiert wird.

Auf einer allgemeinen Modellierungsebene bezeichnet man eine Klasse als abstrakt, wenn sie keine direkten Ausprägungen besitzt. Die Modellierungsklasse “Lebewesen” ist ein Beispiel. Sie ist abstrakt, da es lediglich von Unterklassen wie “Mensch” oder “Katze” konkrete Exemplare gibt (dies geschieht natürlich unter der Annahme, daß man die Klassen “Mensch” und “Katze” nicht selbst weiter klassifizieren muß, was zu weiteren Unterklassen führen würde und die beiden genannten Klassen abstrakt machen würde).

In der Programmierung bezeichnet man eine Klasse als abstrakt, wenn ihre Implementierung nicht vollständig ist, sondern vielmehr ein Gerüst für ihre Unterklassen darstellt. Man setzt abstrakte Klassen methodisch als ein Gerüst ein, daß von konkreten Unterklassen, welche die Implementierung vervollständigen, ausgefüllt wird. Wie weiter unten erläutert wird, setzt man dies mithilfe von klassenbauminternen Vererbungsschnittstellen (“protected”-Schnittstellen) um. Da die Implementierung einer abstrakten Klasse nicht vollständig ist, kann es von ihr auch keine direkten Exemplare geben.

Im engeren Java-Sinne ist eine Klasse abstrakt, wenn sie mittels des Schlüsselworts “abstract” als solche gekennzeichnet wird; dies hat zur Folge, daß sie nicht instantiiert werden kann. Ähnliches kann man auch dadurch erreichen, daß man keine öffentlichen Konstruktoren anbietet. Eine Klasse ist konkret, wenn man sie instantiieren kann, es also Exemplare von ihr gibt. Im Java-Sinne darf sie somit nicht als abstrakt deklariert sein und muß auch mindestens einen öffentlich zugänglichen Konstruktor anbieten.

Die Definition von Namensobjekten reicht natürlich nicht aus, um Objekte unter einem Namen ablegen und wieder hervorholen zu können. Es fehlt uns noch der eigentliche Namensdienst. Glücklicherweise wird seine Schnittstelle bereits in der OMG-COSS definiert, so daß wir sie praktisch nur noch abzuschreiben brauchen:

```
public interface NamingContext
{
  public void bind(Name name, Object object);
  public void rebind(Name name, Object object);
  public void unbind(Name name);
  public Object resolve(Name name);
  public boolean contains(Name name);
}
```

Diese Schnittstelle ermöglicht es einem Klienten, Objekte unter einem Namen abzulegen (bind(Name, Object)) und später unter dem verwendeten Namen auch wieder zu erfragen (resolve(Name)). Man kann Objekte auch umbenennen (rebind(Name, Object)) sowie wieder entfernen (unbind(Name)). Schließlich ist es möglich, zuerst einmal nachzufragen, ob ein Objekt beim Namensdienst überhaupt bekannt ist (contains(Name)). Im zweiten Artikel werden wir sehen, warum der Namensdienst nicht NamingService, sondern Namenskontext (Namenskontext) heißt, wenn wir nämlich das Kompositummuster zur rekursiven Schachtelung von Namenskontexten verwenden.

Möchte man ein frisch erzeugtes Anwendungsobjekt (zum Beispiel einen Browser) unter einem Namen (“KundenBrowser”) bei einem Namensdienst ablegen, so schreibt man in etwa folgenden Code:

```
Browser browser = new BrowserImpl();
Name browserName = new StringNameImpl(“KundenBrowser”);
LokaleDienste.gibNamensDienst().bind(browserName, browser);
...
```

Zu einem späteren Zeitpunkt kann man dann unter dem Namen “KundenBrowser” das Browserobjekt wieder hervorkramen:

```
Name browserName = new StringNameImpl(“KundenBrowser”);
Object object = LokaleDienste.gibNamensDienst().resolve(browserName);
Browser browser = (Browser) object;
...
```

Die letzte Zeile macht deutlich, daß man von einem Namensdienst wie er hier definiert und verwendet wird, immer nur Objekte unter der Schnittstelle Object bekommt, und somit für die Typsicherheit selbst sorgen muß (der Downcast zu Browser sollte nicht fehlschlagen).

Wie implementiert man nun NamingContext? Das Herzstück einer Implementierung muß offenkundig eine Abbildung von Namen auf Objekt(-referenzen) sein. Diese Abbildung kann aber sehr unterschiedlich implementiert werden. Die naheliegendste und auch einfachste Implementierung ist die Verwendung einer Hash-Tabelle für den assoziativen Lookup der Objektreferenz zu einem Namen. Dies funktioniert genau dann gut, wenn alle Namen und Objekte im Hauptspeicher gehalten werden sollen (und können!). Ist dies nicht der Fall, so muß auf andere Implementierungen der Abbildung ausgewichen werden.

CORBA NamingService-Implementierungen zum Beispiel verwalten die Abbildung von Namen nach Objekten in Dateien, nicht zuletzt aus dem Grund, daß die Abbildung nach erneutem Starten des Namensdienstes wieder zur Verfügung stehen muß und nicht verloren gegangen sein darf. Das Heraussuchen einer dateibasierten Repräsentation einer Objektreferenz und das Zurückgeben der tatsächlichen Objektreferenz basiert offenkundig auf ganz anderen Implementierungsmechanismen als der einfache Lookup in einer Hash-Tabelle.

Die Hauptspeichervariante ist schnell, funktioniert aber nur bis zu mittelgroßen Objektmengen. Die Dateivariante ist langsam (zumindest ohne Caching), dafür aber sicher und auch für große Objektmengen geeignet. Somit gibt es mal wieder keine bessere oder schlechtere Implementierung, sondern nur für den gegebenen Zweck angemessene Implementierungen.

Wie können wir jetzt die unterschiedlichen Implementierungen möglichst einfach machen, also eine höchstmögliche Codewiederverwendung erreichen? Die Lösung liegt in der angemessenen Verwendung abstrakter Klassen, welche gemeinsame Aspekte der unterschiedlichen Implentierungsklassen implementieren, aber die unterschiedlichen Aspekte offenlassen.

Dies führt uns im Beispiel zum in Abbildung 2 dargestellten Klassenbau.

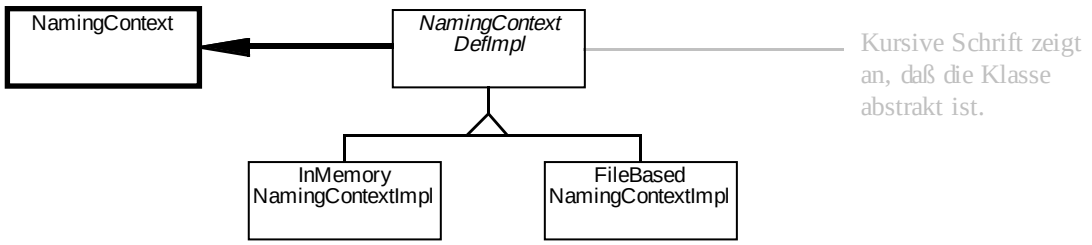

Abbildung 2: Die Schnittstelle von NamingContext mit zwei Implementierungsklassen, deren Gemeinsamkeiten als abstrakte Oberklasse herausfaktorisiert wurden.

Die abstrakte Klasse NamingContextDefImpl implementiert die gemeinsamen Aspekte ihrer vorweg gedachten Unterklassen (DefImpl steht für Default-Implementierung). Dazu verwendet es Konzepte wie Fabrik- und Schablonenmethoden [3], gemeinhin auch als Prinzip der schmalen Vererbungsschnittstelle bekannt (narrow inheritance interface principle). NamingContextDefImpl sieht folgendermaßen aus:

```
public abstract class NamingContextDefImpl
  implements NamingContext
{

protected NamingContextDefImpl()
{
  // da die Klasse abstrakt ist, sollte es auch
  // keine oeffentlichen Konstruktoren geben
}

public void bind(Name name, Object object)
{
  if ( ! contains(name) ) doBind(name, object);
}

public void rebind(Name name, Object object)
{
```

```
  if ( ! contains(name) ) return;
  doUnbind(name);
  doBind(name, object);
}

... und die anderen Operationen

// hier jetzt die Vererbungsschnittstelle
protected abstract void doBind(Name name, Object object);
protected abstract void doUnbind(Name name);
protected abstract void doContains(Name name);

// Das ist es! Kein Implementierungszustand,
// der Unterklassen unnoetig belasten wuerde

}
```

NamingContextDefImpl ist als abstrakte Oberklasse für das Geerbt-werden von noch zu definierenden Unterklassen vorbereitet. Auf der einen Seite implementiert sie bereits eigene Funktionalität, so zum Beispiel die sichtbaren Operationen von NamingContext, tut dies aber gleichzeitig nicht vollständig, sondern delegiert genau jenen Teil an Unterklassen weiter, der von Unterklasse zu Unterklasse variiert. Dieser variable Teil sind im Beispiel die in NamingContextDefImpl definierten abstrakten Operationen doBind, doUnbind and doContains.

Diese Operationen werden jetzt in Unterklassen auf spezifische Weise implementiert. InMemoryNamingContextImpl verwendet zum Beispiel eine Hash-Tabelle:

```
public class InMemoryNamingContextImpl
  extends NamingContextDefImpl
{

public InMemoryContextImpl()
{
  // oeffentlicher Konstruktor, da instantiierbar
}

protected void doBind(Name name, Object object)
{
  table.put(name, object);
}

protected void doUnbind(Name name)
{
  table.remove(name);
}

protected boolean doContains(Name name)
{
  table.contains(name);
}

// Hier endlich mal konkreter Zustand
protected Hashtable table;

}
```

Die Operationen doBind, doUnbind und doContains werden in der Klasse FileBasedNamingContextImpl natürlich ganz anders implementiert. Die Kunst beim Entwurf der abstrakten Oberklasse NamingContextDefImpl besteht darin, die gemeinsame für Unterklassen notwendige Vererbungsschnittstelle zu finden, auf deren Basis NamingContextDefImpl implementiert werden kann. Im Beispiel ist dies gelungen, so daß wir ein hohes Maß an Codewiederverwendung für die Unterklassen erreicht haben, ohne daß wir an Flexibilität verloren hätten.

Natürlich ist es denkbar, daß man auf eine ganz andersartige Implementierung von NamingContext stoßen kann, so daß NamingContextDefImpl nicht wiederverwendet werden kann. Gerade aber dafür gibt es ja die Unterscheidung von Schnittstelle und Implementierung: In einem solchen Fall implementiert diese andersartige Implementierungsklasse (vielleicht TotallyUnrelatedNamingContextImpl) NamingContext direkt, ohne von NamingContextDefImpl zu erben. Warum auch, schließlich kann sie nichts von deren Code gebrauchen.

5 Zusammenfassung

Dieser Artikel hat aufgezeigt, wie man Schnittstellen und Klassen in der Modellierung und Implementierung von Java-basierten Systemen unterscheidet und konstruktiv einsetzen kann. Dieser Artikel hat sich dabei mit den Grundlagen beschäftigt, nämlich der Trennung von Schnittstelle und Implementierung sowie der Verwendung abstrakter und konkreter Klassen. Diese Grundlagen wurden an einem Beispiel, der Implementierung eines einfachen Namensdienstes, illustriert.

Der zweite Artikel, der in der nächsten Java-Spektrum Ausgabe erscheint, befaßt sich mit Vererbung auf Schnittstellen- und Klassenseite. Er diskutiert den Einsatz abstrakter und konkreter Klassen im Rahmen der Klassenvererbung. Besonderes Gewicht wird auf den sinnvollen Einsatz von Codefaktorisierung zur Maximierung der Codewiederverwendung gelegt.

Abschließend möchte ich meinen Kollegen Frank Fröse, Erich Gamma und Kai-Uwe Mätzel für die Diskussion und ihr Feedback zu diesem Artikel danken.

Literaturverweise

[1] NeXTStep. Object-oriented Programming and the Objective C Language. NeXTStep developer's library. NeXT Computer Inc., 1993.

[2] OMG. *CORBA Object Services Specification.* Framingham, MA: Object Management Group, 1997.

[3] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley Deutschland, 1995. (Dies ist die Übersetzung von Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*).

[4] Dirk Bäumer und Dirk Riehle. „Product Trader". *Pattern Languages of Program Design 3.* Addison Wesley, 1997. Kapitel 4.

Zum Autor

Dipl.-Inform. Dirk Riehle arbeitet am Ubilab, dem Informatik-Forschungslabor der UBS (Union Bank of Switzerland). Er ist Autor zahlreicher Fachartikel zu den Themen Objektorientierung und Entwurfsmuster. Außerdem ist er Übersetzer des Buchs *Design Patterns* sowie Autor des frisch bei Addison-Wesley Deutschland erschienenen Buchs *Entwurfsmuster für Softwarewerkzeuge*.

E-Mail: Dirk.Riehle@ubs.com or riehle@acm.org